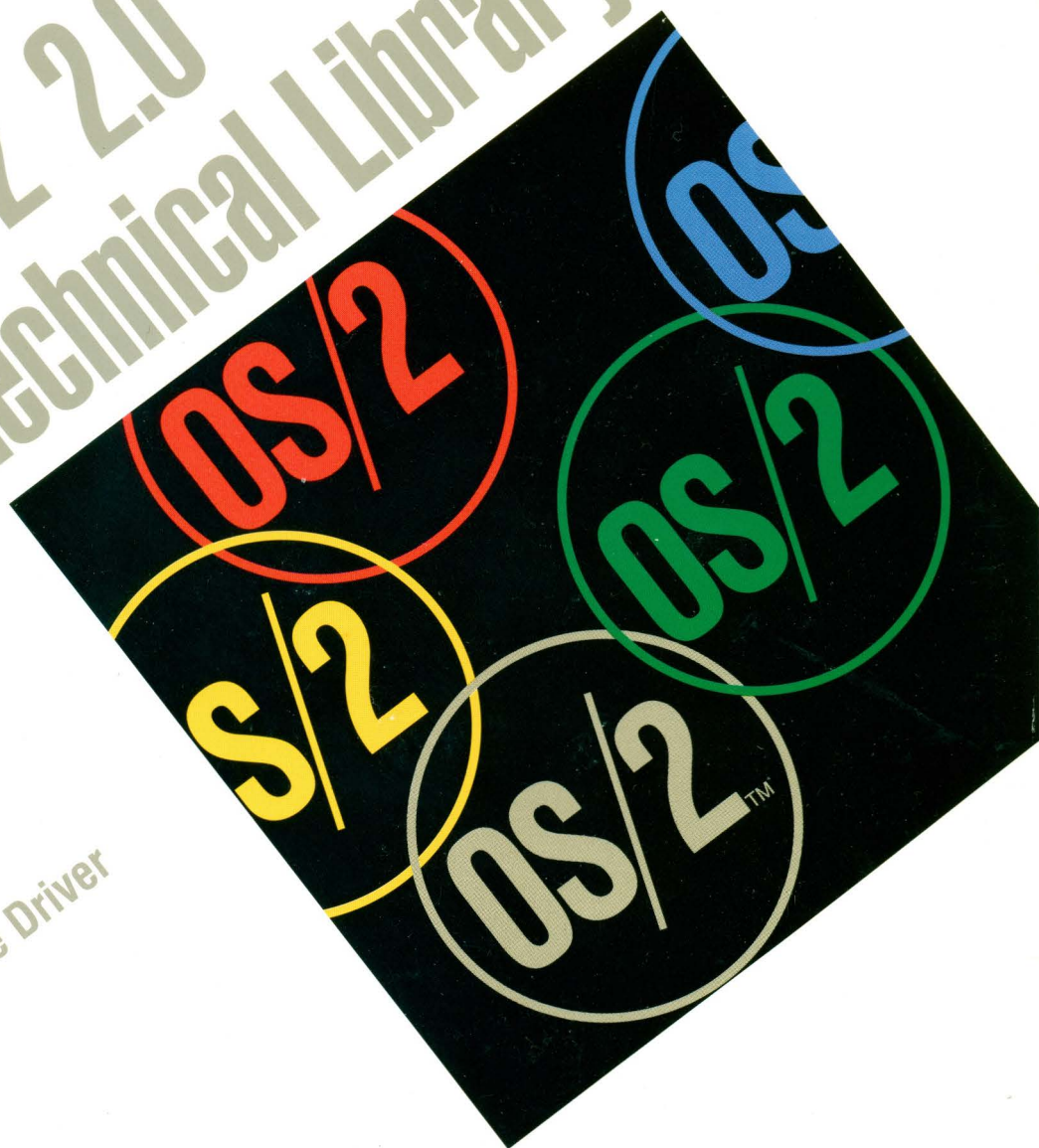
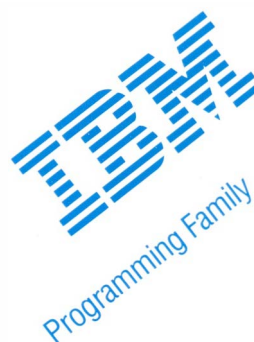


OS/2 2.0 Technical Library



Physical Device Driver
Reference

Version 2.00



OS/2 2.0 Technical Library

**Physical Device Driver
Reference**

Version 2.00



Programming Family

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xiii.

First Edition (March 1992)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "© (your company name) (year) All Rights Reserved."

© Copyright International Business Machines Corporation 1992. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	xiii
Trademarks	xiii
Double-Byte Character Set (DBCS)	xiii

About This Book	xv
------------------------------	----

Part 1. Overview

Chapter 1. Introduction	1-1
I/O Requests from Applications	1-1
Types of OS/2 Device Drivers	1-4
Physical Device Drivers	1-4
Virtual Device Drivers	1-4
Presentation Drivers	1-4
Using the 80386 Microprocessor	1-5
Chapter 2. Physical Device Driver Overview	2-1
Block and Character Device Drivers	2-1
Application I/O to Devices	2-1
I/O Support for OS/2 Applications	2-2
I/O Support for the DOS Session	2-3

Part 2. Development Considerations

Chapter 3. Physical Device Driver Architecture and Structure	3-1
OS/2 Device Driver Contexts	3-1
Physical Device Driver Architecture	3-1
Physical Device Driver Program Model	3-2
Physical Device Driver Header	3-2
Physical Device Driver Components	3-5
Building a Physical Device Driver	3-6
Chapter 4. OS/2 Physical Device Driver Operations	4-1
Physical Device Driver Operations	4-1
Physical Device Driver Initialization	4-2
Hardware Interrupt Management	4-4
Rules for Sharing Interrupt Levels	4-5
DEINSTALL Considerations	4-6
DevHelp Services	4-6
Chapter 5. OS/2 Physical Device Driver Design Issues	5-1
Resource Management	5-1
Request Packet Queue Management	5-1
Memory Management	5-2
Semaphore Management	5-2
Character Queue Management	5-3
Requesting OS/2 Services	5-3
Limiting the Number of Nested Interrupts	5-4
Device Monitor Support in Character Device Drivers	5-4
Device Support	5-6
Converting DOS Session Addresses Within IOCTL Requests	5-6

Inter-Device Driver Communication	5-7
Chapter 6. Character Device Monitors	6-1
Monitoring Character Device Data Streams	6-2
Device Monitor Support Limitations	6-2
The OS/2 Monitor Mechanism	6-4
Character Device Monitor Process	6-5
Character Device Driver with Monitor Support	6-6
Character Device Driver and Monitors	6-7
OS/2 Monitor Functions	6-9
DosMonRead	6-11
DosMonWrite	6-12
DosMonClose	6-13
Guidelines for a Character Device Monitor	6-14
Monitor Buffers	6-14
Monitor Data Records	6-15
Positioning of Monitors in a Monitor Chain	6-15
Monitor Thread Priorities	6-16
Special Considerations for Character Device Monitors	6-17
Performance	6-17
Monitor Termination	6-18
Well-Behaved Monitor Applications	6-19
Monitor Problems and Solutions	6-22
Type-Ahead Characters	6-22
Redirecting Data to Another Device	6-24
Providing Monitor Support in a Character Device Driver	6-25
MonitorCreate	6-25
Register	6-26
MonWrite	6-27
MonFlush	6-28
DeRegister	6-28
Guidelines for a Character Device Driver	6-29
Buffer Requirements	6-29
Code Requirements	6-30
Special Considerations for Character Device Drivers	6-31
Device-Specific Monitor Information	6-31
Performance	6-32
Device Driver Problems	6-32
Sample Programs	6-33
A Character Device Driver (DEMODB.ASM)	6-33
A Character Device Monitor (MONITOR.ASM)	6-33
Chapter 7. Installation of External Loadable Device Drivers	7-1
Device Driver Profile	7-1

Part 3. OS/2 2.0 Physical Device Drivers

Chapter 8. Physical ASYNC (RS232-C) Communications Device Driver	8-1
Hardware Support	8-1
IBM PS/2 Micro Channel Adapter Support	8-1
AT Bus Adapter Support	8-2
Attachment Support	8-2
RS232-C Interface	8-2
Hardware Support for Extended Hardware Buffering	8-3
INS 8250, INS 8250-B Considerations	8-3

ASYNC (RS232-C) Device Driver Features	8-4
Error Alert Generation	8-7
States of the ASYNC Device Driver	8-8
Enhanced Mode	8-15
DMA Channel Arbitration	8-15
DMA Receive Operation	8-16
RS232-C Enabling Characteristics in Enhanced Mode	8-19
Input Modem Control Signals in Enhanced Mode	8-20
Logical Flow Control (XON/XOFF) in Enhanced Mode	8-20
Bit Rate Considerations	8-21
Control Panel Considerations	8-22
Reserved Device Names (COM1-n)	8-22
IBM PS/2 (with Micro Channel) Considerations for COM1-4	8-22
Initialization/Resource Management	8-22
Data Translation/Monitor Support/Spooler Support	8-24
ASYNC Communication Device Driver Interfaces	8-24
File System Requests	8-25
Access Authorization	8-27
ASYNC (RS232-C) Generic IOCTL Command Summary	8-28
DOS Session Considerations/Restrictions	8-29
Control Panel Considerations	8-29
Performance	8-29
 Chapter 9. Physical CLOCK\$ Device Driver	 9-1
 Chapter 10. Physical Fixed Disk and Diskette Device Driver	 10-1
Configuration	10-1
Performance Considerations	10-1
EXTDSKDD.SYS Support	10-1
DOS Extended Volume Architecture	10-2
Installing Block Devices in the DOS Extended Partition	10-3
Creating Block Devices in the Extended DOS Partition	10-4
Deleting Block Devices in the Extended DOS Partition	10-4
BPB and Get Device Parameters for Extended Volumes	10-7
Category 8 Generic IOCTL Commands	10-7
Category 9 Generic IOCTL Commands	10-8
Type 6 Partition	10-8
Type 7 Partition	10-10
 Chapter 11. Physical Keyboard Device Driver	 11-1
Keyboard Input (KBD\$)	11-1
Keyboard Initialization	11-2
Keyboard Run Time Operation	11-3
Keystroke Monitors	11-3
Special Key Processing	11-7
 Chapter 12. Physical Mouse Device Driver	 12-1
Generic Pointing Device Support	12-1
High-Level Design	12-1
Application Level Interface	12-2
Physical Mouse Device Driver Considerations	12-2
Adding Support for a Unique Pointing Device	12-2
MOUSE.SYS IDC Interface	12-2
Device-Dependent Device Driver IDC Interface	12-6
 Chapter 13. Physical Parallel Port Device Driver	 13-1

Overview	13-1
Replacing the Parallel Port Device Driver	13-2
Parallel Port IRQ Performance	13-2
DMA Parallel Port Support	13-3
Code Page Support	13-3
Character Monitor Performance	13-6
Parallel Port IRQ Timeout Processing	13-6
Parallel Port Device Driver Interfaces	13-7
Request Packet Interface	13-7
INT 21H Interface	13-7
Generic IOCtl Interface	13-8
Hardware Interrupt (8259) Interface	13-8
Monitor Dispatcher Notification Interface	13-9
Parallel Port Monitor Buffer Command	13-10
Chapter 14. Physical Video Device Drivers	14-1
Video Device Handler Identification	14-1
Video Device Chaining	14-2
A VGA and 8514A Scenario	14-2
Primary Display Identification	14-3
WrtToScrn/Panic Write Support	14-3
Running System Installation	14-3
Using Loadable Device Drivers	14-3
Video Device Handler Interfaces	14-4
DevEnable	14-5
InitEnable	14-7
Text Buffer Update	14-8
Initialize Environment	14-11
Save Environment	14-12
Restore Environment	14-14
Query Config Info	14-15
Query DBCS Display Info	14-16
Query Color Lookup Table	14-17
Set Color Lookup Table	14-18
Query Cursor Info	14-19
Set Cursor Info	14-20
Query Font	14-21
Set Font	14-22
Query Mode	14-23
Set Mode	14-24
Query Palette Registers	14-25
Set Palette Registers	14-26
Query Physical Buffer	14-27
Free Physical Buffer	14-28
Query Variable Info	14-29
Set Variable Info	14-31
Terminate Environment	14-33
Print Screen	14-34
Write TTY	14-35
Query LVB Info	14-36
Level 0 Physical Device Driver Interfaces	14-37
INIT (00H) - Initialize the Device	14-37
OPEN (0DH) - Open the Device	14-37
GENERIC IOCtl (10H) - Send I/O Requests to the Device	14-37
Physical Device Driver Initialization	14-38
EGA.SYS and INT 2F Screen Switch Notification	14-39

EGA.SYS Device Driver	14-39
EGA Register Interface	14-39
Function F0 — Read One Register	14-42
Function F1 — Write One Register	14-43
Function F2 — Read Register Range	14-44
Function F3 — Write Register Range	14-45
Function F4 — Read Register Set	14-46
Function F5 — Write Register Set	14-48
Function F6 — Revert to Default Registers	14-49
Function F7 — Define Default Register Table	14-50
Function F8 — Read Default Register Table	14-51
Function FA — Interrogate Driver	14-52
INT 2F Screen Switch Notification	14-54
Using Extended Screen and Keyboard Control (ANSI.SYS, ANSICALL)	14-54
Limitations/Restrictions	14-54
Control Sequence Syntax	14-54
Cursor Control Sequences	14-55
Erasing	14-57
Controlling Display Mode	14-57
Keyboard Key Reassignment	14-58

Part 4. Reference Material

Chapter 15. Physical Device Driver Strategy Commands	15-1
Request Packets	15-1
Summary of Strategy Commands	15-4
0H / INIT	15-5
1H / MEDIA CHECK	15-7
2H / BUILD BPB	15-9
4H, 8H, 9H / READ/WRITE/WRITE WITH VERIFY	15-11
5H / NONDESTRUCTIVE READ NO WAIT	15-12
6H, AH / INPUT OR OUTPUT STATUS	15-13
7H, BH / FLUSH INPUT OR OUTPUT	15-14
DH, EH / OPEN OR CLOSE DEVICE	15-15
FH / REMOVABLE MEDIA	15-16
10H / GENERIC IOCTL	15-17
11H / RESET MEDIA	15-18
12H, 13H / GET OR SET LOGICAL DRIVE MAP	15-19
14H / DEINSTALL	15-20
16H / PARTITIONABLE FIXED DISKS	15-21
17H / GET FIXED DISK/LOGICAL UNIT MAP	15-22
1CH / SHUTDOWN	15-23
1DH / GET DRIVER CAPABILITIES	15-24
Chapter 16. Extended Device Driver Interface	16-1
Disk Device Driver Architecture	16-1
Standard OS/2 Strategy Routine	16-2
Extended Strategy Routine	16-2
Sorting and Priority	16-2
Request Management	16-2
Removable Media	16-3
Devices Not Capable of Scatter/Gather	16-3
Identifying Extended Device Drivers and Capabilities	16-3
GET DRIVER CAPABILITIES Command	16-3
Request Lists and Request Control	16-6

Extended Commands	16-8
Request Control Functions	16-13
Chapter 17. Device Helper (DevHlp) Services	17-1
Using DevHlp Services	17-1
Calling the DevHlp Interface Routine	17-1
Register Usage	17-1
State of the Interrupt Flag	17-1
Constant Definitions	17-1
16:16 Virtual Address Conversion	17-1
New DevHlp Services	17-2
DevHlp Services and Function Codes	17-3
DevHlp Services and Device Contexts	17-6
Related DevHlp Services	17-9
ABIOSCall	17-12
ABIOSCommonEntry	17-13
AllocateCtxHook	17-14
AllocGDTSelector	17-15
AllocPhys	17-16
AllocReqPacket	17-17
ArmCtxHook	17-18
AttachDD	17-19
Beep	17-20
Block	17-21
CloseEventSem	17-23
DeRegister	17-24
DevDone	17-25
SAVE_MESSAGE	17-26
DynamicAPI	17-27
EOI	17-28
FreeCtxHook	17-29
FreeGDTSelector	17-30
FreeLIDEntry	17-31
FreePhys	17-32
FreeReqPacket	17-33
GetDescInfo	17-34
GetDeviceBlock	17-35
GetDOSVar	17-36
GetLIDEntry	17-39
InternalError	17-40
LinToGDTSelector	17-41
LinToPageList	17-42
Lock	17-43
MonFlush	17-45
MonitorCreate	17-46
MonWrite	17-48
OpenEventSem	17-49
PageListToGDTSelector	17-50
PageListToLin	17-52
PhysToGDTSel	17-54
PhysToGDTSelector	17-55
PhysToUVirt	17-56
PhysToVirt	17-57
PostEventSem	17-60
ProtToReal	17-61
PullParticular	17-62

PullReqPacket	17-63
PushReqPacket	17-64
QueueFlush	17-65
QueueInit	17-66
QueueRead	17-67
QueueWrite	17-68
RealToProt	17-69
Register	17-70
RegisterBeep	17-71
RegisterPDD	17-72
RegisterStackUsage	17-73
RegisterTmrDD	17-74
ResetEventSem	17-75
ResetTimer	17-76
ROMCriticalSection	17-77
Run	17-78
SchedClockAddr	17-79
SemClear	17-80
SemHandle	17-81
SemRequest	17-83
SendEvent	17-84
SetIRQ	17-85
SetROMVector	17-86
SetTimer	17-87
SortReqPacket	17-88
TCYield	17-89
TickCount	17-90
Unlock	17-91
UnPhysToVirt	17-92
UnSetIRQ	17-93
VerifyAccess	17-94
VideoPause	17-95
VirtToLin	17-96
VirtToPhys	17-97
VMAlloc	17-98
VMFree	17-100
VMGlobalToProcess	17-101
VMLock	17-103
VMProcessToGlobal	17-105
VMSetMem	17-107
VMUnlock	17-108
Yield	17-109
Chapter 18. Generic IOCTL Commands	18-1
Generic IOCTL Function Table	18-3
Category 1 ASYNC (RS232-C) Control IOCTL Commands	18-7
Function 41H — Set Bit Rate	18-8
Function 42H — Set Line Characteristics	18-9
Function 43H — Extended Set Bit Rate	18-11
Function 44H — Transmit Byte Immediate	18-13
Function 45H — Set Break OFF	18-15
Function 46H — Set Modem Control Signals	18-16
Function 47H — Behave as if XOFF Received	18-18
Function 48H — Behave as if XON Received	18-19
Function 4BH — Set Break ON	18-20
Function 53H — Set DCB Parameters	18-21

Function 54H – Set Enhanced Mode Parameters	18-36
Function 61H – Query Bit Rate	18-39
Function 62H – Query Line Characteristics	18-40
Function 63H – Extended Query Bit Rate	18-41
Function 64H – Query COM Status	18-42
Function 65H – Query Transmit Data Status	18-43
Function 66H – Query Modem Output Signals	18-44
Function 67H – Query Modem Input Signals	18-45
Function 68H – Query Number of Characters in Receive Queue	18-46
Function 69H – Query Number of Characters in Transmit Queue	18-47
Function 6DH – Query COM Error	18-48
Function 72H – Query COM Event Information	18-49
Function 73H – Query DCB Parameters	18-50
Function 74H – Query Enhanced Mode Parameters	18-53
Category 3 IOCTL Commands	18-54
Category 3 Pointer Draw Control IOCTL Command	18-55
Function 72H – Query Pointer Draw Address	18-56
Category 3 Video Control IOCTL Command	18-57
Function 73H – Initialize Call Vector Table	18-58
Category 3 Screen Control IOCTL Commands	18-59
Function 70H – Allocate a Selector	18-60
Function 71H – Deallocate a Selector	18-61
Function 74H – BIOS Pass-Through	18-62
Function 75H – Allocate a Selector with Offset	18-63
Function 76H – Allocate a Selector with Background Validation	18-64
Category 4 Keyboard Control IOCTL Commands	18-65
Function 50H – Set Code Page	18-66
Function 51H – Set Input Mode	18-77
Function 52H – Set Interim Character Flags	18-78
Function 53H – Set Shift State	18-79
Function 54H – Set Typematic Rate and Delay	18-80
Function 56H – Set Session Manager Hot Key	18-81
Function 57H – Set KCB	18-83
Function 58H – Set Code Page Number	18-84
Function 59H – Set Read/Pek Notification	18-85
Function 5AH – Alter Keyboard LEDs	18-86
Function 5CH – Set NLS and Custom Code Page	18-87
Function 5DH – Create a Logical Keyboard	18-88
Function 5EH – Destroy a Logical Keyboard	18-89
Function 71H – Query Input Mode	18-90
Function 72H – Query Interim Character Flags	18-91
Function 73H – Query Shift State	18-92
Function 74H – Read Character Data Records	18-93
Function 75H – Peek Character Data Record	18-95
Function 76H – Query Session Manager Hot Key	18-97
Function 77H – Query Keyboard Type	18-99
Function 78H – Query Code Page Number	18-100
Function 79H – Translate Scan Code to ASCII	18-101
Function 7AH – Query Keyboard Hardware ID	18-103
Function 7BH – Query Keyboard Code Page Information	18-104
Category 5 Parallel Port Control IOCTL Commands	18-105
Function 42H – Set Frame Control	18-106
Function 44H – Set Infinite Retry	18-107
Function 46H – Initialize Parallel Port	18-108
Function 48H – Activate Font	18-109
Function 4DH – Set Print-Job Title	18-110

Function 4EH – Set Parallel Port IRQ Time-Out Value	18-111
Function 62H – Query Frame Control	18-112
Function 64H – Query Infinite Retry	18-113
Function 66H – Query Parallel Port Status	18-114
Function 69H – Query Active Font	18-115
Function 6AH – Verify Font	18-116
Function 6EH – Query Parallel Port IRQ Time-Out Value	18-117
Category 7 Mouse Control IOCTL Commands	18-118
Function 51H – Notification of Display Mode Change	18-119
Function 53H – Reassign Mouse Scaling Factors	18-122
Function 54H – Assign New Mouse Event Mask	18-123
Function 56H – Set Pointer Shape	18-124
Function 57H – Unmark Collision Area	18-126
Function 58H – Mark Collision Area	18-127
Function 59H – Specify/Replace Pointer Position	18-128
Function 5AH – Set OS/2 Mode Pointer Draw Device Driver Address	18-129
Function 5CH – Set Physical Mouse Device Driver Status Flags	18-131
Function 5DH – Notification of Mode Switch Completion	18-132
Function 60H – Query Number of Mouse Buttons Supported	18-133
Function 61H – Query Mouse Device Motion Sensitivity	18-134
Function 62H – Query Physical Mouse Driver Status Flags	18-135
Function 63H – Read Mouse Event Queue	18-136
Function 64H – Query Event Queue Status	18-137
Function 65H – Query Mouse Event Mask	18-138
Function 66H – Query Mouse Scaling Factors	18-139
Function 67H – Query Pointer Screen Position	18-140
Function 68H – Query Pointer Shape	18-141
Function 6AH – Query Physical Mouse Device Driver Version Number	18-142
Function 6BH – Query Pointing Device ID	18-143
Category 8 Logical Disk Control IOCTL Commands	18-144
Function 00H – Lock Drive	18-145
Function 01H – Unlock Drive	18-146
Function 02H – Redetermine Media	18-147
Function 03H – Set Logical Map	18-148
Function 04H – Begin Format	18-149
Function 20H – Block Removable	18-150
Function 21H – Query Logical Map	18-151
Function 43H – Set Device Parameters	18-152
Functions 44H, 64H, 65H – Write/Read/Verify Track	18-154
Function 45H – Format and Verify Track	18-156
Function 60H – Query Media Sense	18-158
Function 63H – Query Device Parameters	18-159
Category 9 Physical Disk Control IOCTL Commands	18-161
Function 00H – Lock Physical Drive	18-162
Function 01H – Unlock Physical Drive	18-163
Functions 44H, 64H, 65H – Write/Read/Verify	18-164
Function 63H – Query Physical Device Parameters	18-166
Category 10 Character Device Monitor IOCTL Command	18-167
Function 40H – Register Monitor	18-168
Category 11 General Device Control IOCTL Commands	18-170
Function 01H – Flush Input Buffer	18-171
Function 02H – Flush Output Buffer	18-172
Function 41H – System Notifications for Physical Device Drivers	18-173
Function 60H – Query Monitor Support	18-177

Appendixes

Appendix A. Running OS/2 Version 1.3 16-Bit Physical Device Drivers on OS/2 2.0	A-1
Use of Physical Addresses - PhysToUVirt	A-1
Direct Call to Physical Device Drivers	A-1
Direct Writing of GDT Selectors	A-1
Physical Device Drivers that Need Real Mode	A-1
Physical Device Drivers Support of DOS Applications	A-1
 Appendix B. Using Advanced BIOS	B-1
Device Driver Data Segment	B-1
Obtaining a Logical ID	B-1
Calling Advanced BIOS Services	B-2
Mapping Device Names to LID	B-2
Handling ABIOS Requests	B-3
Writing a Physical Device Driver Using Advanced BIOS	B-3
Spurious Interrupts	B-6
Address Conversion Using DevHlp Services	B-7
 Glossary	X-1
 Index	X-11

Notices

Trademarks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

IBM	OS/2	PS/2
IBM PS/2	AT	Personal System/2
IBM Personal System/2	PC AT	Micro Channel
IBM Personal Computer	Personal Computer AT	
IBM Personal Computer AT	Presentation Manager	

The following terms, denoted by a double asterisk (**) in this publication, are trademarks of other companies as follows:

Intel	Intel Corporation
Logitech	Logitech, Inc.
PC Mouse	Metagraphics/Mouse Systems
Microsoft	Microsoft Corporation
XENIX	Microsoft Corporation

Double-Byte Character Set (DBCS)

Throughout this publication, there are references to specific values for character strings. These values are for the Single-Byte Character Set (SBCS). When using the Double-Byte Character Set, note that one DBCS character equals two SBCS characters.

About This Book

The *OS/2 2.0 Physical Device Driver Reference* defines what a physical device driver is and how it operates. In addition, a description of the types of physical device drivers, their interfaces, and the available system services is provided. System and application programmers can use the information found in this book to write their own physical device drivers and subsystems.

Knowledge of at least one programming language that is used for writing OS/2 applications is necessary, and the programmer must be familiar with the workings of the OS/2 operating system and the Presentation Manager interface. The programming concepts that should be understood before developing applications to run on OS/2 2.0 are found in the *OS/2 2.0 Application Design Guide*.

"OS/2," as used in this book, refers to Version 2.00 of the OS/2 operating system unless stated otherwise.

This book consists of four parts: An introductory overview of device drivers (Part 1); an information section on writing physical device drivers (Part 2); a breakdown of the different types of OS/2 physical device drivers (Part 3); and a detailed reference section (Part 4). Also included are two appendixes covering OS/2 1.3 16-bit physical device drivers running on OS/2 2.0, and Advanced BIOS. A more detailed description of the contents follows:

Part 1. Overview

Chapter 1. Introduction

This chapter contains a general description of OS/2 physical device drivers, presentation drivers and virtual device drivers, explaining their differences and how they fit into the operating system.

Chapter 2. Physical Device Driver Overview

This chapter contains a detailed description of OS/2 physical device drivers including the types of drivers, installation, and I/O support for applications.

Part 2. Development Considerations

Chapter 3. Physical Device Driver Architecture and Structure

This chapter contains a description of the physical device driver architecture, component structure, and driver contexts.

Chapter 4. OS/2 Physical Device Driver Operations

This chapter contains a breakdown of physical device driver operations, including physical device driver initialization and hardware interrupt management.

Chapter 5. OS/2 Physical Device Driver Design Issues

This chapter contains design considerations for physical device drivers and PDD/VDD pairs, a description of the types of communication between device drivers, and how to build a physical device driver.

Chapter 6. Character Device Monitors

This chapter contains a description of character device monitors, their processes, support, and monitor functions.

Chapter 7. Installation of External Loadable Device Drivers

This chapter contains information on how to install external loadable device drivers, and a description and sample of a device driver profile.

Part 3. OS/2 2.0 Physical Device Drivers (Each chapter below describes a type of physical device driver):

Chapter 8. Physical ASYNC (RS232-C) Communications Device Driver

Chapter 9. Physical CLOCK\$ Device Driver

Chapter 10. Physical Fixed Disk and Diskette Device Driver

Chapter 11. Physical Keyboard Device Driver

Chapter 12. Physical Mouse Device Driver

Chapter 13. Physical Parallel Port Device Driver

Chapter 14. Physical Video Device Drivers

Part 4. Reference Material

Chapter 15. Physical Device Driver Strategy Commands

This chapter contains a description of the commands that the device driver strategy routine must support.

Chapter 16. Extended Device Driver Interface

This chapter contains a description of the extended device driver interface used for the service of fixed disks devices.

Chapter 17. Device Helper (DevHlp) Services

This chapter contains descriptions, listed alphabetically, of the kernel functions (DevHlp services) that are available to physical device drivers.

Chapter 18. Generic IOCTL Commands

This chapter contains a description of each of the IOCTL commands listed by category and function, and guidelines for using the IOPL code segments.

Appendixes

Appendix A. Running OS/2 Version 1.3 16-Bit Physical Device Drivers on OS/2 2.0

This appendix describes the problems and solutions of running a 16-bit physical device driver on the OS/2 2.0 operating system.

Appendix B. Using Advanced BIOS

This appendix contains information on how the physical device driver uses the Advanced BIOS methods, including obtaining a Logical ID, handling ABIOS requests, and spurious interrupts.

A glossary and an index are included in the back of the book.

Part 1. Overview

Chapter 1. Introduction

Physical device drivers act as an interface between the OS/2 2.0 operating system (or its applications) and the physical device. They resolve device-independent input and output (I/O) requests from the operating system and its applications with the device-dependent physical attributes of the device. The types of physical device drivers shipped with OS/2 2.0 include support for:

- Asynchronous Communication (RS-232C)
- System Clock
- Fixed Disk and Diskette
- Keyboard
- Mouse
- Parallel Port Printer
- Video.

OS/2 device drivers execute entirely in protect mode. The physical device drivers service requests from the OS/2 operating system, OS/2 applications, and DOS applications. *Virtual device drivers* service I/O requests for DOS applications that directly access BIOS or system hardware. For more information on virtual device drivers, see the *OS/2 2.0 Virtual Device Driver Reference*.

This book provides a description of the OS/2 physical device drivers, including:

- Physical device driver structure and architecture
- Design considerations, including performance and nested interrupts
- Interfaces to a physical device driver, including the request packet and generic IOCTL interfaces
- Subsystem interfaces to a physical device driver
- OS/2 services available for physical device drivers, that is, Device Helper (DevHlp) services
- Inter-device driver communication (device drivers calling other device drivers).

I/O Requests from Applications

Request packets are the primary method of communication between the OS/2 operating system and a device driver. The OS/2-provided physical device drivers service requests to both the DOS Sessions and the OS/2 operating system. Where appropriate, these device drivers provide a queued request interface rather than the serial request design of DOS device drivers. OS/2 physical device drivers support multi-tasking.

OS/2 physical device driver functions are not usually directly invoked by applications. The OS/2 File System and Device Manager provide the application interface. The generic IOCTL interface does allow an application or subsystem to send device-specific control commands to the physical device driver. The IOCTL interface for OS/2 applications or subsystems is the `DosDevIOCtl` function call. The IOCTL interface for DOS applications is `INT 21H`.

The OS/2 operating system receives I/O requests from applications and sends data in the form of request packets to the physical device driver. The physical device driver then translates the request packet into a sequence of I/O operations that operate the device. An application request for I/O reaches a physical device driver through one or more of the following interfaces:

I/O Subsystems: Functions that insulate an application from managing device I/O.

The File System: Dynamic link functions that an application can use to redirect I/O.

* Trademark of the IBM Corporation

Introduction

The IOCTL Interface: The function DosDevIOctl allows an application to send device-specific commands to a physical device driver. See the *OS/2 2.0 Control Program Programming Reference* for a description of the DosDevIOctl function. See Chapter 18, "Generic IOCTL Commands" for a description of the IOCTL interface for system physical device drivers.

Character Device Monitors: Dynamic link functions that allow an application to view, insert, or modify data passing to, or from, a device.

Figure 1-1 illustrates how applications run in an OS/2 full screen session:

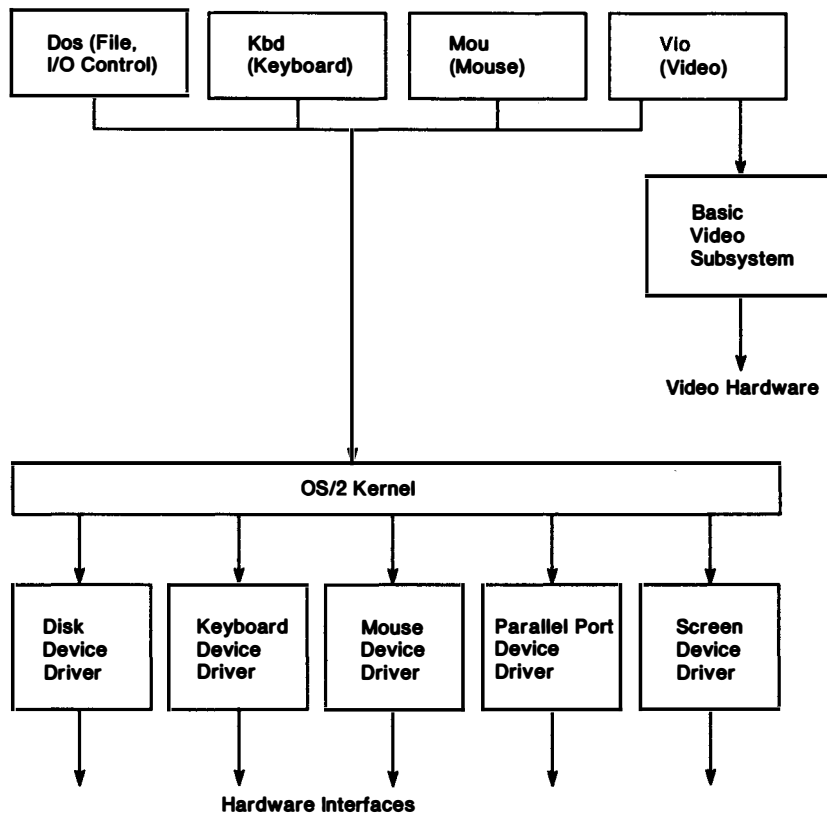


Figure 1-1. Applications Running in an OS/2 Full Screen Session

Figure 1-2 illustrates how applications run in a Presentation Manager session:

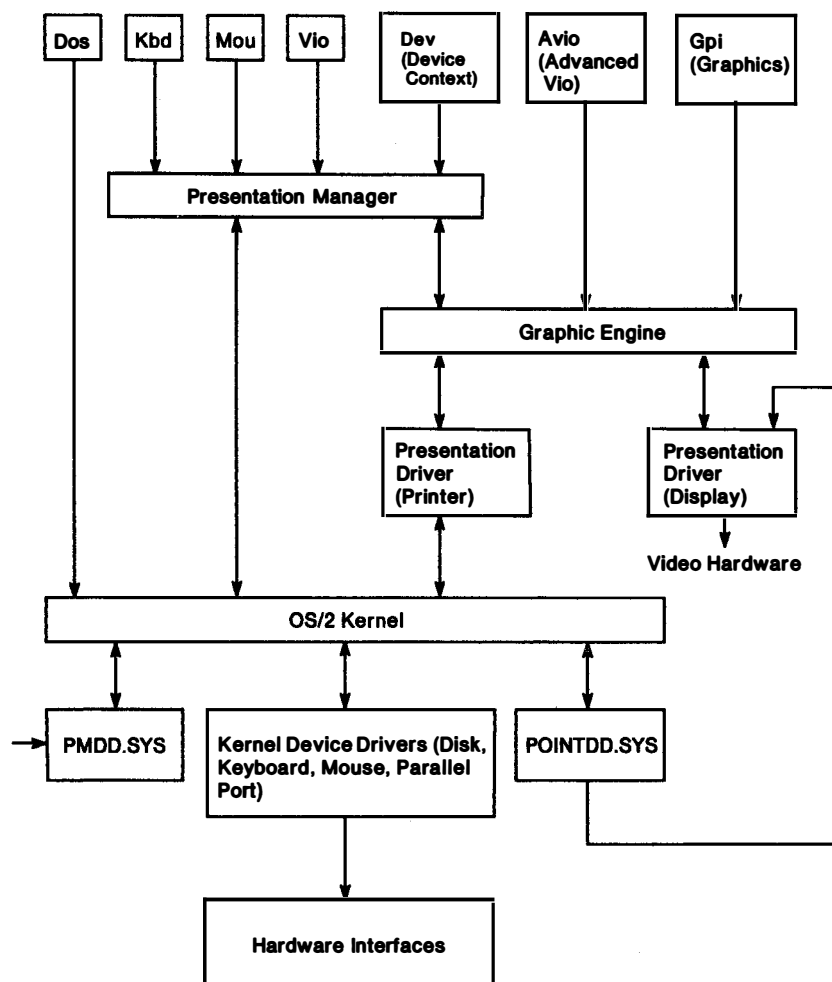


Figure 1-2. Applications Running in a Presentation Manager Session

Types of OS/2 Device Drivers

Three types of device drivers are used in OS/2 2.0:

- Physical device drivers
- Virtual device drivers
- Presentation drivers.

Physical Device Drivers

Standard I/O devices are supported by base physical device drivers that are part of the OS/2 operating system. Additional, or replacement, physical device drivers can be loaded to extend the control provided by the base device drivers or to support nonstandard I/O devices. Typical examples of these loadable device drivers are ANSI.SYS and ANSI.DLL, which are loaded by `DEVICE =` statements in `CONFIG.SYS` and provide additional functions on the interfaces to the screen and keyboard. Physical device drivers are initialized at Ring 3 and operate at Ring 0, the most privileged level of the operating system.

Virtual Device Drivers

The virtual device driver is an installable module responsible for virtualizing a particular piece of hardware and associated ROM BIOS in the manner expected by a DOS application. This device driver achieves virtualization by emulating I/O port and device memory operations. Virtual device drivers are 32-bit device drivers, which operate at Ring 0. To achieve a certain level of hardware independence, a virtual device driver usually communicates with a physical device driver in order to interact with hardware.

Further information on virtual device drivers, virtual device driver interfaces (including detailed descriptions of the calling conventions), and the system services available to these drivers is found in the *OS/2 2.0 Virtual Device Driver Reference*.

Presentation Drivers

The Presentation Manager* I/O interface for output devices is a high-level interface. This interface is similar to the API call interface, which uses the program stack to communicate with, or pass parameters to, the presentation drivers. These drivers are special purpose I/O routines operating with I/O privilege at privilege level 2 (Ring 2) or level 3 (Ring 3). Their main function is to process function calls made by the Presentation Manager interface on behalf of Presentation Manager applications. Hardcopy presentation drivers communicate with OS/2 device drivers through the file system emulation functions. Display presentation drivers interface directly with the hardware.

Presentation drivers are dynamic link library modules that are supplied as files and identified by the extension, `DRV`. When the Presentation Manager interface is initialized, the display presentation driver is loaded and enabled automatically. Other presentation drivers (for example, the hardcopy presentation drivers) are loaded and enabled when an application calls the `DevOpenDC` function to open the device.

Presentation drivers service requests only from applications running in Presentation Manager sessions in the OS/2 mode. Output data and requests for information are passed to the presentation driver as function calls to the presentation driver interface. The definition of the call interface is given in terms of the codes and data passed to the presentation driver interface through the program stack.

Header and include files are shipped with the OS/2 operating system to provide support for building presentation drivers that are written in C or assembler language. These files contain function prototypes, defined values, and data structures used by the various functions. Further information on presentation

* Trademark of the IBM Corporation

drivers, presentation driver interfaces (including detailed descriptions of the calling conventions), and the system services available to these drivers is found in the *OS/2 2.0 Presentation Driver Reference*.

Using the 80386 Microprocessor

OS/2 2.0 is a 32-bit operating system that runs on an 80386 (or higher) microprocessor. The OS/2 operating system utilizes the full capabilities of the 386 processor, including:

- 32-bit addressing and calculations
- Flat (nonsegmented) memory model
- Protect mode execution (DOS Sessions execute in V8086 Emulation mode as opposed to real mode).

Chapter 2. Physical Device Driver Overview

Physical device drivers written for DOS are synchronous and often noninterrupt driven. Because DOS is a single-task operating system, this presents no problem. A program cannot proceed until the I/O has been completed, so it is acceptable for the physical device driver to hold the Central Processing Unit (CPU) until the I/O is complete.

OS/2 2.0 is a multi-tasking operating system, which can assign the CPU to tasks that are not waiting for I/O. Therefore, a basic characteristic of the OS/2 physical device driver is the support of a multi-tasking environment. Physical device drivers written for the OS/2 operating system must surrender the CPU while they are waiting for I/O completion. Consequently, OS/2 physical device drivers must be interrupt-driven.

Block and Character Device Drivers

There are two types of physical device drivers:

- Character device drivers
- Block device drivers.

Character device drivers manage I/O on character-oriented devices. These devices perform I/O on a character-by-character basis. A character device has a name like SCREEN\$, KBD\$, LPT1, or COM1. A character device driver can support more than one device by having multiple linked device headers where each header indicates a different name.

Block device drivers support block-oriented devices. These devices perform I/O on a block of data, typically through Direct Memory Access (DMA). Applications request I/O on block-oriented devices through the file system. Consequently, block device drivers do not have names. Instead, a block device driver is assigned one or more drive letters. Since a block device driver can support multiple devices (or units), a drive letter is assigned for each device (or unit). A block device driver specifies the number of devices that it supports, when it is called to initialize. Refer to the physical device driver strategy command, "OH / INIT" on page 15-5. The order in which the block device drivers appear in the CONFIG.SYS configuration file (through the DEVICE= command) determines the order in which they receive drive letters.

Application I/O to Devices

The primary interface to a physical device driver is the OS/2 request packet interface. File I/O and IOCTL requests from OS/2 applications and INT 21H requests from DOS applications running in a DOS Session are translated into request packets that are sent to the physical device driver for processing.

An OS/2 physical device driver is the standard interface to a device. OS/2 device drivers manipulate the devices on behalf of all applications and subsystems. However, when the physical device driver interface is insufficient, both OS/2 and DOS applications can:

- Access a device directly by sending data to, or receiving data from, a specific port address through I/O instructions. Notice that 32-bit protect-mode applications cannot issue I/O. Only 16-bit protect-mode applications can issue I/O from the Ring 2 code segment.
- Manipulate the state of the hardware interrupts (disable/enable interrupts) through CLI/STI instructions.

Any DOS application can directly access a device from its code segment. OS/2 applications and subsystems can directly access a device only through IOPL code segments. For DOS applications, I/O direct to the hardware is dependent on which virtual device driver is installed. The virtual device driver can register and hook to the particular port for I/O.

Figure 2-1 illustrates how DOS and OS/2 applications communicate with a device.

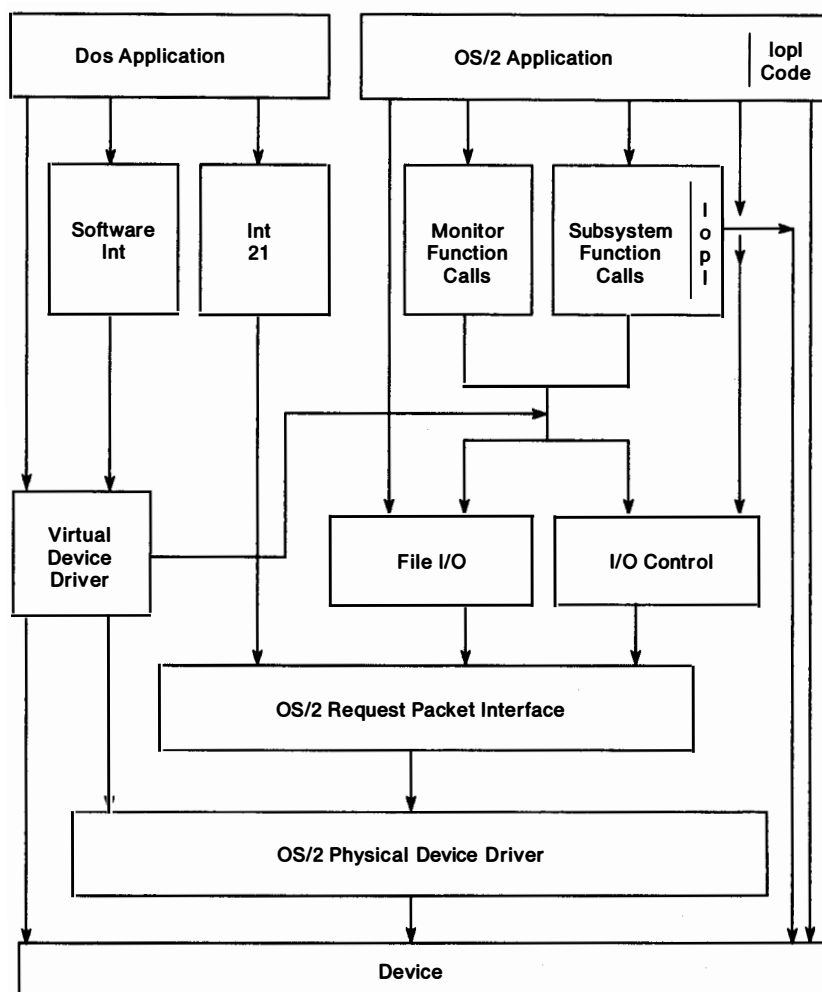


Figure 2-1. Application I/O to a Device Driver

I/O Support for OS/2 Applications

An OS/2 application can perform I/O requests by using the interfaces that access the physical device drivers. These interfaces are described in the following table:

Interface	Example	Advantage
File System	DOS dynamic link functions	An application can redirect the I/O.
Subsystem	VIO, KBD, MOU functions	Insulates an application from managing device I/O.
IOCtl	DosDevIOCtl commands	A subsystem or application uses IOCtl commands to send device-specific control commands to device drivers.
Character Device Monitors	OS/2 dynamic link monitor functions	An application can directly insert, view or modify data passing to or from a device.

OS/2 Interfaces that Access Device Drivers

The *File System interface* consists of the file I/O functions. Both OS/2 and DOS applications use the DOS dynamic link calls for file I/O. DOS applications running in a DOS Session issue INT 21H file I/O function calls, which are passed to the OS/2 file systems. These file I/O functions are used primarily to perform I/O on block devices, such as fixed disks. Some file I/O functions can be used to perform I/O on character devices. These functions include:

- DosOpen
- DosClose
- DosRead
- DosReadAsync
- DosWrite
- DosWriteAsync.

The advantage of using the file I/O functions to perform I/O on a character device is that the application can redirect the input/output. The application performs I/O to a handle (file or device) that it obtained by opening the named resource passed to it.

A *Subsystem interface* shields an application from having to manage device input/output. The file system is an example of a subsystem. It allows the application to view a file as a logical sequence of sectors, thus shielding the application from having to manage the physical locations of the data on the disk media.

The *IOctl interface* is the mechanism that an application or subsystem uses to send device-specific control commands to the device driver. DosDevIOctl is the IOctl mechanism used for new applications. The INT 21H IOctl request is used for DOS applications. I/O commands can be sent to both block and character device drivers. The application or subsystem must first obtain the device handle by performing an OPEN on the device name.

Character Device Monitors are supported by the OS/2 2.0 operating system. These monitors provide a mechanism for applications or subsystems to monitor all characters passing to, or from, a character device driver. This mechanism allows one or more registered processes to remove, insert, or modify data passing through a character device.

A physical device driver is the standard way to provide service to and from OS/2 applications. However, occasionally an application written under an earlier version of OS/2 might directly access a device through service routines defined in the IOPL code segments of the application.

I/O Support for the DOS Session

A device driver is responsible for managing input and output for its device. Because system resources must be accessible to both DOS applications and OS/2 applications, OS/2 device drivers for system resources come in physical device driver/virtual device driver pairs (PDD/VDD). (Physical device drivers have the additional capability of communicating with virtual device drivers.) PDD/VDD pairs are provided for the disk, keyboard, screen, mouse, and parallel port.

Device I/O in a DOS Session is performed in one of three ways:

- DOS INT 21H interface
- ROM BIOS interface
- Direct access to the device.

I/O that uses the *DOS INT 21H interface* is passed to the OS/2 File System and transformed into a request packet, which is received by the physical device driver. I/O that uses the OS/2 dynamic link functions is similarly transformed into a request packet. OS/2 physical device drivers service INT 21H functions that create standard I/O request packets. This level of support exists in all OS/2 system device drivers.

I/O that uses the *ROM BIOS interface* is handled by the virtual device driver. The OS/2 virtual device driver must intercept the ROM BIOS software interrupt and pass the request along to its associated physical device driver. The virtual device driver and physical device driver serialize access to the device by using semaphores to indicate when the device is busy with a request (and consequently cannot accept or tolerate a request from ROM BIOS). *Direct access* for standard system resources, such as RS232-ASYNC communication, disk, keyboard, mouse, parallel port, video, numeric coprocessor, and ROM BIOS, are virtualized by virtual device drivers provided by the OS/2 operating system.

Note: Virtual device drivers are not normally required for feature hardware (for example, scanners, FAX, or MIDI) where the feature is operated only from a single DOS Session.

Part 2. Development Considerations

Chapter 3. Physical Device Driver Architecture and Structure

The architecture and structure of a physical device driver differs considerably, depending on whether the physical device driver is physical or virtual. A physical device driver is considered a *true* device driver in the sense that it has a unique and rigid file structure, and interfaces directly with the hardware. A virtual device driver is essentially a dynamic link library, which generally does not interface directly with the hardware.

OS/2 Device Driver Contexts

There are three contexts (modes) in which OS/2 device drivers operate:

- **Kernel Mode**

The OS/2 kernel calls the physical device driver strategy routine for task-time operations; that is, it will execute as a thread within a process. The strategy routine will not be preempted by a task switch but may be interrupted by incoming hardware interrupts. Kernel mode is also referred to as *task context*.

- **Interrupt Mode**

The OS/2 kernel calls the physical device driver interrupt-time components, the hardware interrupt handler, and the timer handler. *Interrupt time* is a generic term that refers to executing code as a result of an interrupt; the thread of execution does not belong to a process. The hardware interrupt handler and the timer handler do not execute code as a thread belonging to a thread-specific process. The thread of execution results from a hardware interrupt.

- **INIT Mode**

The physical device driver strategy routine is called with a request packet containing the INIT command. The initialization code runs at Ring 3 with I/O privilege. A limited set of dynamic link function calls are available for use, as well as a portion of the device helper (DevHlp) function calls. For more information refer to "Physical Device Driver Initialization" on page 4-2.

Physical Device Driver Architecture

The data segment must be the first segment after the EXE file header. Since the device header must be located at the beginning of the file, this allows it to be located immediately after the EXE file header.

Note: Multi-segmented device drivers that support character device monitors must be organized so that the monitor notification routine resides in the first code segment or the fixed memory.

The file image of a physical device driver is shown in Figure 3-1 on page 3-2.

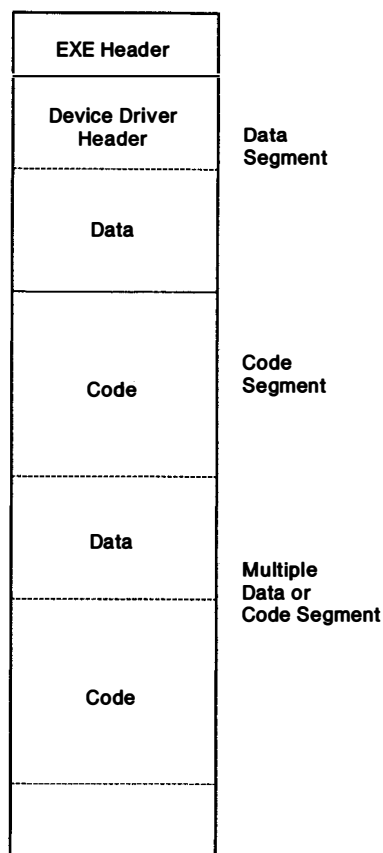


Figure 3-1. File Image of a Physical Device Driver

Physical Device Driver Program Model

An OS/2 device driver may have multiple code and data segments. The first segment in the physical device driver must be a data segment, and the second must be a code segment. These two segments are fixed in memory. By default, only the first two segments are kept in the system after device driver initialization. OS/2 assumes that every other segment is needed for initialization only, unless it is marked as permanent in the header of the EXE file.

A permanent segment remains after initialization and is assumed to be movable and swappable, but not discardable. A physical device driver can lock a permanent segment using the appropriate Device Helper service, if it is necessary to prevent it from being swapped out or moved. IOPL is available at privilege level 3 during device driver installation. The physical device driver developer uses the SEGMENTS directive and its IOPL option in the linker DEF file to mark those segments that are to be kept after initialization. Since the first two segments are always kept, they do not have to be marked in this fashion.

Note: Unlike the OS/2 Version 1.x operating systems, the OS/2 2.0 device drivers always execute in protect mode.

The physical device driver executable image does not contain a stack segment. A stack is provided by the system. However, at initialization, a device driver may indicate to the system its stack usage by calling the RegisterStackUsage device helper routine for each IRQ it services.

Physical Device Driver Header

The first data segment of a physical device driver must contain a device header as the first item. The structure of a physical device driver header is shown in Table 3-1 on page 3-3.

Field	Length
Pointer to Next Device Header	DWORD
Device Attribute	WORD
Offset to Strategy Routine	WORD
Offset to IDC Entry Point	WORD
Name or Units	8 BYTES
Reserved	8 BYTES
Capabilities Bit Strip (only in function level 3 device drivers)	DWORD

Table 3-1. Physical Device Driver Header Structure

Pointer to Next Device Header: This pointer is set by OS/2 2.0 at the time the physical device driver is loaded. This field should be set to -1 .

Note: For a character device driver that supports multiple devices, the data segment contains a device header for each device. These headers must be linked together and the last header must be set to -1 . The first WORD is an offset; the second WORD is the segment.

Device Attribute: Describes the characteristics of the physical device driver to the system. The format of the OS/2 device attribute field is:

C	I	I	S	O	///		///	///	///	C	N	S	K
H	D	B	H	P	///	LEVEL	///	///	///	L	U	C	B
R	C	M	R	N	///		///	///	///	K	L	R	D

Figure 3-2. Device Attribute Field

The attributes are:

Bit 15 Set, if the physical device driver operates in character mode. Bit 15 is the device type bit. Use bit 15 to tell the system if the device driver is a block or character device. For block device drivers, bit 15 is 0. For character device drivers, bit 15 is 1.

Bit 14 Set, if the physical device driver participates in inter-device driver communication (IDC). Bit 14 is the IDC bit and indicates that the offset to the IDC entry point in the physical device driver header is set.

Bit 13 Set, if non-IBM block format (block device drivers only). Set for output-until-busy support (character device drivers only).

For block device drivers, bit 13 indicates the method the physical device driver uses to determine the media type. If a block device driver uses information in the BIOS Parameter Block (BPB) to determine the media type; bit 13 should be set to 1. If the physical device driver uses the media descriptor byte to determine the media type; bit 13 should be 0.

Bit 12 Set, if it supports shared-device access-checking (character devices). Bit 12 is the shared bit. It is set if the device name is not to be protected by sharer. Bit 12 has no meaning for block device drivers and must be 0.

If clear (default), file system sharing rules do not apply to the device, and it is the responsibility of the physical device driver to provide contention control.

If set, file system sharing rules apply to the device, just like they apply to any other file system name. In addition, any given physical device can have only one logical name. (Devices cannot have aliases.)

Bit 11 Set, if it supports removable media (block devices) or device open/close (character devices). For block device drivers, bit 11 is the removable media bit. If set, this bit indicates that the physical device driver handles removable media.

For character device drivers, bit 11 is the open/close bit. If set, this bit indicates that the physical device driver must receive OPEN and CLOSE request packets.

Bit 10 Reserved = 0.

Bits 9–7 Bits 7-9 indicate the physical device driver function level, where:

001 = OS/2 device driver

010 = Supports DosDevIOCtl2 and SHUTDOWN request packets

011 = Uses a Capabilities Bit Strip in the header.

Bit 6 Reserved = 0.

Bit 5 Reserved = 0.

Bit 4 Reserved = 0.

Bit 3 Set, if CLOCK device. Bit 3 is the clock device bit. It is used by a character device driver to indicate the system clock device.

Bit 2 Set, if NULL device. Bit 2 is the NULL attribute bit. It is used by character devices only. Bit 2 is used to tell the OS/2 operating system if the character device driver is a NULL device. Although there is a NULL device attribute bit, the NULL device cannot be reassigned. (This attribute exists so the operating system can tell if the NULL device is being used.)

Bit 1 Set, if standard output device (STDOUT). For character devices, bits 1 and 0 are the standard input or standard output bits. These bits are used to tell the OS/2 operating system if the character device driver is the new standard input or standard output device.

Bit 0 Set, if standard input device (STDIN). See above.

Offset to Strategy Routine: Contains the offset from the start of the code segment to the strategy entry point. The OS/2 operating system uses this offset to call the strategy routine. The pointer is a WORD value contained in the device header.

Offset to the IDC Entry Point: The offset from the start of the code segment of the physical device driver to the entry point, which is callable by other device drivers.

Name or Units: Contains the name of a character device supported by the character device driver or the number of units supported by the block device driver.

For a character device driver, the name of the device must be uppercase ASCII characters and left-justified with the remaining space set to blanks. The device name is used by applications to identify the device for I/O. A character device driver should consider the following rule when selecting a physical device driver name:

A device name takes precedence over a filename in a DosOpen function. This means that files cannot have the same name as a character device. The DosOpen function always opens the device, rather than the file.

Note: To avoid such conflicts with filenames, a character device driver should choose a character string with some unusual character such as a \$ sign.

Capabilities Bit Strip: Function Level 3 device drivers have an additional DWORD field appended to the device header. Each bit of this new field is a flag indicating whether or not a particular feature is supported. The flags in this field are:

Bit 0 Set to 1, if DosDevIOCtl2 request packets are supported, and if shutdown support is provided.

- Bit 1** For character device drivers, this bit is set to 1 (if memory addressing above 16MB is supported, that is, support for full 32-bit memory addressability versus 24-bit memory addressability). For block device drivers, this bit is reserved and must be set to 0.
- Bits 2** Set to 1, if the physical device driver supports parallel ports.
- Bits 3-31** Reserved. Must be set to 0.

Note: Physical device drivers at Function Levels 1 and 2 do not have or need this field in their headers.

Physical Device Driver Components

The components of a physical device driver are illustrated below:

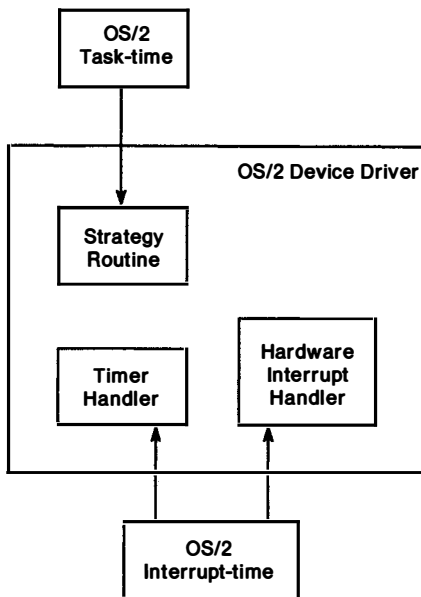


Figure 3-3. Components of a Physical Device Driver

Deciding what components to include in a physical device driver depends on two things, the complexity of the task, and how the application performs I/O to its device. A physical device driver is comprised of the strategy routine, the hardware interrupt handler, and the timer handler:

- **Strategy Routine (Required)**

The strategy routine is called to handle I/O requests through a request packet interface with the OS/2 kernel. The pointer to the request packet is in ES:BX.

The strategy routine executes at task-time as a result of an application I/O request. Application I/O requests can come from new OS/2 applications active in the protect mode of the processor and from DOS applications active in a DOS Session (through a virtual device driver).

The strategy routine follows the far call/return model. When it completes processing, it issues a far return to the kernel. By convention, OS/2 preserves any registers used by the strategy routine.

The request packet interface to device drivers is described in Chapter 15, "Physical Device Driver Strategy Commands."

- **Hardware Interrupt Handler (Recommended)**

The hardware interrupt handler is called as a result of a hardware interrupt and executes at interrupt time. The interrupt handler follows the FAR CALL/RETURN model. When it completes processing, it issues a FAR RETURN to the OS/2 operating system. The handler must clear or set the Carry Flag (CF)

to indicate whether it owns the hardware interrupt. By convention, OS/2 2.0 preserves any registers used by the hardware interrupt handler.

- **Timer Handler (Optional)**

The timer handler is called as a result of a periodic clock tick and executes at interrupt time. This handler manages timeouts and is similar to the INT 1CH user time feature of ROM BIOS.

The timer handler follows the FAR CALL/RETURN model. When it has completed processing, it issues a FAR return to OS/2. The handler must save and restore any registers it uses.

Building a Physical Device Driver

Writing a physical device driver allows new or optional devices to be installed without modifying the operating system. To create a physical device driver that the OS/2 operating system can install:

1. Create the DATA and CODE segments. A physical device driver can have multiple code and data segments. See "Physical Device Driver Program Model" on page 3-2 and Figure 3-1 on page 3-2. The first segment after the .EXE file header must be the first data segment (default). Because a physical device driver can have multiple code and data segments, it can make FAR calls to other code segments.

Note: Since OS/2 installs the physical device driver anywhere in memory, care must be taken in any memory references. Do not expect that the physical device driver will be loaded at the same place every time.

By default, only the first data and code segments remain in memory after initialization of the physical device driver. All other segments are assumed to be initialization code, unless otherwise designated. Memory used by these segments is returned to the system.

2. Define the physical device driver header. The physical device driver header must be the first data in the first data segment and must be located immediately after the .EXE file header. The physical device driver header defines the characteristics of a physical device driver for the OS/2 operating system. See "Physical Device Driver Header" on page 3-2 for a detailed description of each device driver header field.
3. Define a strategy routine and interrupt handler routine in the code segments.
4. Define the code and data segments and their attributes in the linker DEF file (module definition file). The SEGMENTS directive and its IOPL option in the DEF file indicate which segments are to be permanent, that is, remain in memory after initialization.
5. Link the code and data segments as a library.
6. Include a DEVICE= statement for the physical device driver in the CONFIG.SYS file so the physical device driver is installed when the system is initialized.
7. Prepare a Device Driver Profile (see "Device Driver Profile" on page 7-1) to include on a device support diskette so that users can install the physical device driver by using the DDINSTALL utility.

Chapter 4. OS/2 Physical Device Driver Operations

Physical device driver operations consist of the activities and interactions between the strategy routine and the hardware interrupt handler in the processing of an I/O request.

Physical Device Driver Operations

The handling of an I/O request begins with OS/2 calling the strategy routine entry point with a request packet. The strategy routine checks the validity of the I/O request. If valid, the strategy routine might place the request on a work queue for the device, using the DevHlp services for request queue management.

If the device is currently idle, the strategy routine starts the request at the device. The strategy routine may then wait for the device driver interrupt by suspending its thread of execution with the DevHlp service, BLOCK.

When the device interrupt occurs, the hardware interrupt handler checks the request to see if it has been completed. If the request has not been completed, the hardware interrupt handler continues the request at the device. If the request has been completed, the hardware interrupt handler sets the return status in the request packet. The hardware interrupt handler may remove the completed request packet from the work queue and start the next request at the device. If the strategy routine is waiting for the device interrupt (which is blocked), the hardware interrupt handler can initiate the strategy routine with the DevHlp, RUN.

The strategy routine queues the requests and initiates only the I/O, if the device has been inactive. The hardware interrupt handler starts requests as they reach the head of the work queue. The thread that is running when the physical device driver determines that a particular request is complete, might not necessarily be the same thread in which the device driver had received the request. This is particularly true for the interrupt-time components of the physical device driver. For example, the address of a user buffer passed to the physical device driver when the request was issued belongs to a specific Local Descriptor Table (LDT), which might not be the current LDT when the request ends. The device driver can accommodate this by storing the buffer address as a 32-bit physical address.

The physical device driver strategy routine is called by the OS/2 operating system with a pointer to the request packet. Any addresses passed in the request packets for Read/Write requests are passed as 32-bit physical addresses (normalized). Therefore, the physical device driver does not need to lock or convert the addresses into physical addresses. The device driver only needs to lock addresses that it receives from a source other than the OS/2 operating system, as in the case of a process passing an address by way of a generic IOCTL.

The multi-tasking environment dictates that the components of the device driver must be capable of handling requests simultaneously. This means that the components of the physical device driver must relinquish execution whenever possible. The physical device driver relinquishes control at task-time by blocking, yielding, or referencing a segment which had been swapped out. The OS/2 operating system does not preempt a thread in the physical device driver. However, once the physical device driver releases its execution, the operating system calls the physical device driver with a new request. Once the strategy routine blocks, yields, or references a swapped-out segment, its thread of execution is called with a new request under the context of a different thread.

While the strategy routine assumes that it will not be preempted by other task-time instances, it must protect itself against its own interrupt-time components. It should disable interrupts, when checking if the device is active, and when examining the device queue. The interrupt-time components are preempted only by other higher priority interrupts.

Physical Device Driver Initialization

Physical device driver initialization occurs during system initialization. During system initialization, base device drivers are pre-loaded with the operating system. Installable device drivers are loaded during CONFIG.SYS processing, by way of the DEVICE = configuration command. After a physical device driver has been loaded, it will be called at its strategy routine entry point with the request packet for the INIT command.

OS/2 device drivers have the following characteristics at INIT-time. They can:

- Initialize in protect mode
- Have IOPL at all times or execute at Ring 3
- Service hardware interrupts
- Use timer services
- Use some OS/2 dynamic link function
- Use BIOS services.

During CONFIG.SYS processing, each DEVICE = command is processed on a first-come first-served basis:

- The DEVICE = device driver file image is loaded into memory.
- The check for DEINSTALL of a previously initialized device driver is performed.
- If DEINSTALL is successful, the newly loaded device driver is initialized.

Installable OS/2 device drivers initialize in protect mode. The device driver strategy routine will run under the thread of the System Initialization Process at Ring 3, with I/O privilege. Because of the special system process, the installable device driver is allowed to make dynamic link function calls only at INIT-time.

Device Driver INIT-Time Function Call Summary: A limited subset of the dynamic link API functions (Dosxxx functions) can be used by the physical device drivers at initialization time. The dynamic link functions available to a physical device driver at initialization time include:

DosBeep	Generate sound from speaker
DosCaseMap	Perform case mapping
DosChgFilePtr	Change (move) file read/write pointer
DosClose	Close file handle
DosDelete	Delete file
DosDevConfig	Get device configuration
DosDevIOctl	I/O control for devices
DosFindClose	Close find handle
DosFindFirst	Find first matching file
DosFindNext	Find next matching file
DosGetEnv	Get address of process environment string
DosGetInfoSeg	Get address of system variables segment
DosGetMessage	System message with variable text
DosOpen	Open file
DosPutMessage	Output message text to indicated handle
DosQCurDir	Query current directory
DosQCurDisk	Query current disk
DosQFileInfo	Query file information
DosQFileMode	Query file mode
DosRead	Read from file
DosSMRegisterDD	Register session switch notification
DosWrite	Synchronous write to file.

For a detailed description of each function, see the *OS/2 2.0 Control Program Programming Reference*.

Note: Because device driver initialization is invoked from the strategy routine, a physical device driver must not issue a DosExit function call. Instead, the physical device driver should return the INIT request packet by setting the return status of the packet, and performing a FAR RET to the kernel.

In addition to the set of dynamic link function calls, certain DevHlp services are available to device drivers at INIT-time. For a list of these DevHlp services, see "DevHlp Services and Function Codes" on page 17-3. For more information on device driver initialization, see the description of the device driver strategy command "OH / INIT" on page 15-5.

Replacing Character Device Drivers: OS/2 character device drivers can be replaced. For system character device drivers, the appropriate bits in the attribute field of the device driver header and the name of the character device driver must match.

When the new device driver is loaded, the attribute field and name are used to determine if the new device driver is attempting to replace a driver already installed. If so, the previously installed device driver is requested to deinstall the indicated device. If this driver refuses the DEINSTALL command, the new device driver is not allowed to initialize. If the previously installed device driver performs the deinstall operation, the new device driver is initialized.

Note: The DEINSTALL command is needed to allow a physical device driver to relinquish its interrupt vectors and its allocated physical memory.

Compatibility with Previous-Level DOS Device Drivers: Not all previous-level (any level below DOS 5.0) DOS device drivers can be allowed to run in the DOS Session. The supported set of previous-level device drivers has the following restrictions:

- Previous-level block device drivers are not permitted in the a DOS Session. Block device drivers must be written to the OS/2 interfaces.
- Only a limited set of previous-level character device drivers can be supported by OS/2. In order to run in a DOS Session, a previous-level character device driver must conform to the following rules:
 - The character device driver cannot have a hardware interrupt handler; its device must be a polled device rather than an interrupt-driven one.
 - The character device driver can be called by the OS/2 operating system in a DOS Session, with all of the packets supported by character devices:
 - 0 – INIT
 - 3 – IOCTL input
 - 4 – INPUT (read)
 - 5 – NON-DESTRUCTIVE INPUT NO WAIT
 - 6 – INPUT STATUS
 - 7 – INPUT FLUSH
 - 8 – OUTPUT
 - 9 – OUTPUT with verify
 - 10 – OUTPUT STATUS
 - 11 – OUTPUT FLUSH
 - 12 – IOCTL output
 - 13 – DEVICE OPEN
 - 14 – DEVICE CLOSE
 - 16 – GENERIC IOCTL

Receipt of these packets is limited by the same requirements on the attribute field of the device header as in DOS 5.0. For example: the IOCTL bit in the device header must be 1 to receive IOCTL requests.

Initialization of Previous-Level DOS Device Drivers: A previous-level DOS character device driver can be initialized to run in a single DOS Session or in all DOS Sessions. A previous-level device driver can be installed for a single DOS Session by using DOS Settings. DOS device drivers can also be installed for all DOS Sessions by specifying the configuration command, `DEVICE =`.

Previous-level device drivers are loaded and initialized in a DOS Session. The rules for replacing these drivers are the same. The replacement is guided by the name and attributes of the physical device driver. The functions that can be performed at initialization are more restrictive than for DOS 5.0. No INT 21H functions can be performed from the physical device driver initialization code.

Hardware Interrupt Management

The hardware interrupt handler is the component of the physical device driver that deals with a hardware interrupt. The handler is called by the OS/2 kernel when the hardware interrupt occurs, and must follow the FAR CALL/RET model. By convention, the hardware interrupt handler does not need to save and restore any registers it uses because this is done by the kernel.

Before the handler can be invoked, its entry point must be registered for a specific hardware interrupt. This may be done during or after device driver initialization with the DevHlp service, `SetIRQ`. Once the call to the DevHlp has been made, the hardware interrupt handler can be invoked.

Interrupt Level Sharing: A hardware interrupt level that is shared among two or more devices is referred to as a *shared interrupt*. Interrupt sharing is an extension of a device's design. A single interrupt level (IRQ) can be shared among two or more devices, if the devices are specifically built for interrupt sharing. A physical device driver cannot share the hardware interrupt level without cooperation from its device. This is true for both edge-triggered and level-sensitive interrupt environments.

Note: Interrupt sharing might not work on all OEM machines that run OS/2 2.0.

In an edge-triggered interrupt environment, an interrupt request is recognized by the 8259 Programmable Interrupt Controller (PIC) as a particular edge transition (like low-to-high) on the hardware interrupt request line. The interrupt request line can remain level without generating another interrupt. The 8259 PIC requires an End-Of-Interrupt (EOI), and another of the same kind of edge transition, to recognize an interrupt on that interrupt level.

In a level-sensitive interrupt environment, an interrupt request is recognized by the 8259 PIC as a particular level on the hardware interrupt request line. The interrupt condition must be removed before the EOI is issued, or else the 8259 continues to generate interrupts for that interrupt level.

Note: The OS/2 operating system supports interrupt sharing on the IBM PS/2* and the EISA machines, which provide a level-sensitive interrupt environment where multiple device drivers can share a particular hardware interrupt.

The basic model for managing a hardware interrupt follows:

1. The physical device driver must register an interrupt handler for a hardware interrupt, specifying whether the physical device driver intends to share the interrupt level.
2. When invoked, the physical device driver interrupt handler tests the device to see if it generated the interrupt.
3. If the device has an interrupt pending or caused a spurious interrupt, the interrupt handler owns the processing of the interrupt. The handler services the device, resets the interrupting condition at the

* Trademark of the IBM Corporation

device, issues the End-Of-Interrupt (EOI) with the DevHlp service EOI, performs a FAR return (RET FAR) indicating that the interrupt handler owned the interrupt, and clears the carry flag (CF=0).

4. If the device does not have an interrupt pending, the interrupt handler does not own processing of the interrupt. The handler must RET FAR indicating that it does not own the interrupt, and set the carry flag (CF=1).

Rules for Sharing Interrupt Levels

All interrupt levels have the potential to be shared. There are some restrictions for permitting two or more device drivers to share an interrupt level, each device driver must adhere to the rules on the following actions:

Interrupt Level Sharing

SYSTEM TIMER RULE: The system timer interrupt level (IRQ 0) cannot be shared.

ILL-BEHAVED DEVICE RULE: An interrupt handler for an ill-behaved device must not share an interrupt level, since an ill-behaved device generates interrupts before its interrupt handler is installed, or cannot be told to stop generating interrupts.

Well-behaved devices do not power-up with interrupts pending, and do not remain active after their handlers have terminated. Also, well-behaved devices do not usually generate spurious interrupts.

SET IRQ RULE: The physical device driver must indicate when signing up for a hardware interrupt level with the DevHlp, SetIRQ, whether it will share the interrupt level.

IRQ ENFORCEMENT RULE: If a physical device driver signs up for a hardware interrupt indicating that it will not share the hardware interrupt, then a subsequent SetIRQ request to share that hardware interrupt is refused. Conversely, if a physical device driver signs up for a hardware interrupt indicating that it will share the hardware interrupt, then a subsequent SetIRQ request to exclusively own the hardware interrupt is refused.

IRQ MASK RULE: The operating system owns the masking of the hardware interrupt at the 8259 interrupt controller, and enables the hardware interrupt when the first interrupt handler signs up for the hardware interrupt. This permits the handler to communicate with its device during initialization.

Processing The Interrupt

STI ENTRY RULE: Interrupt handlers that share interrupts are entered with processor interrupts enabled. This prevents the lockout of higher priority hardware interrupts, because the search for the owner of the current interrupt level takes a variable amount of time. Interrupt handlers that do not share interrupts are entered with processor interrupts disabled.

IRQ OWNERSHIP RULE: The physical device driver interrupt handler, when invoked, must always interrogate its device to see if the device caused the interrupt. If the interrupt handler's device caused the interrupt, then the handler owns the processing of the interrupt and can briefly disable processor interrupts for critical operations. It must issue the EOI as soon as possible and must be aware that once it does so, it could be reentered at its interrupt handler's entry point. If the interrupt handler's device did not cause the interrupt, the handler must not issue an EOI.

INT RETURN RULE: The interrupt handler, after taking the appropriate action in processing the interrupt, must return an indication whether it claimed the interrupt. If the handler owns the interrupt, then it must clear the carry flag (CF=0) and issue a FAR return (FAR RET) when processing is complete. If the handler does not own the interrupt, then it must set the carry flag (CF=1) and issue a FAR RET.

SEARCH RULE: The operating system calls each interrupt handler registered for a particular interrupt level until one of the interrupt handlers claims the interrupt. If no interrupt handler claims the interrupt, then the IRQ is disabled.

EOI RULE: Management of the 8259 interrupt controllers is the responsibility of the operating system. However, the End-Of-Interrupt (EOI) is the responsibility of the interrupt handler. The handler must use the DevHlp EOI service to issue the EOI as soon as possible in the processing of its interrupt. This permits the 8259 interrupt controller to process other interrupt requests at the current interrupt priority, as well as interrupt requests of lower priorities. In a level-sensitive interrupt environment, the EOI must not be issued to the 8259 interrupt controllers until the interrupt condition at the device is removed.

PhysToVirt RULE: Selectors used for PhysToVirt represent a critical resource; an interrupt handler that uses PhysToVirt must not issue the EOI until after it no longer needs the addresses generated by PhysToVirt. Otherwise, the interrupt handler should disable processor interrupts before issuing the EOI. This allows the interrupt handler to use the temporary selectors for its interrupt level without getting another interrupt on its level. However, this does not apply to the selectors obtained by the AllocGDT DevHlp.

POSITION RULE: An interrupt handler that shares an interrupt level must not depend on its position in the list of handlers for that interrupt level.

DEINSTALL Considerations

A single device driver with a single interrupt handler for a particular interrupt level can share its interrupt level among one or more devices that it owns. This case applies to devices of similar or same nature; for example, a printer device driver supporting more than one printer (adapter) on one interrupt level. For this case, the operating system invokes the physical device driver's interrupt handler when the hardware interrupt occurs. The physical device driver's interrupt handler must determine which of its devices caused the interrupt. The handler does not get called for each device it manages.

If the physical device driver is supporting multiple devices on the same hardware interrupt, it only needs to process the first device it discovers causing an interrupt on the interrupt level in question.

Because of the level-sensitive interrupt environment, other devices that are requesting service can cause another interrupt to be generated, and the physical device driver to be re-entered at the same point. Because of this, the physical device driver should strategically place the EOI to allow an orderly processing of any re-entrant interrupt requests.

However, if the physical device driver registers a separate unique interrupt handler entry point for each device it owns, with the interrupt handlers sharing the same interrupt level, the physical device driver must adhere to the interrupt sharing rules. The operating system invokes each interrupt handler, until one handler claims the interrupt. To the operating system, each registered interrupt handler entry point appears as a separate interrupt handler. This allows the physical device driver's interrupt handlers to be called for each device.

The OS/2 operating system allows a maximum of four interrupt handlers to be registered to share one interrupt level at one time. These handlers can be in one device driver or in different device drivers. Each use of the DevHlp, SetIRQ, increases by one the number of registered interrupt handlers for an interrupt level. Each use of the DevHlp, UnSetIRQ, reduces by one the number of registered interrupt handlers for an interrupt level. Refer to "14H / DEINSTALL" on page 15-20 for more detailed information.

DevHlp Services

Some OS/2 services are limited by the context in which the device driver is running. See Chapter 17, "Device Helper (DevHlp) Services" on page 17-1 for a complete list of the DevHlp services, functional descriptions, and limitations.

Chapter 5. OS/2 Physical Device Driver Design Issues

For the best use of the OS/2 operating system, design the physical device driver to conform to these standard system performance guidelines:

- Physical device drivers written for DOS are synchronous and poll their devices. Because DOS is a single-task operating system, a device driver can hold the CPU until I/O is complete. In the OS/2 multi-tasking environment, however, physical device drivers must surrender the CPU while they wait for I/O completion. Consequently, OS/2 physical device drivers must be interrupt driven.
- To provide the best performance when access to the structures or buffers must take place at both task time and interrupt time, locate critical data structures, or data transfer areas in the physical device driver's data segment. For further details, see "Building a Physical Device Driver" on page 3-6.
- Design the physical device driver strategy routine to provide checks to yield the CPU about every 3 milliseconds (ms). If the physical device driver strategy routine does not check to yield the CPU and it executes for periods longer than 3 milliseconds, it could delay dispatching of a process that is ready to run.
- As a guideline, design both the physical device driver strategy routine and the hardware interrupt handler to run with processor interrupts enabled as much as possible. Plan to have the interrupt handler issue the End-Of-Interrupt (EOI) as soon as critical processing is performed. If the physical device driver does processing after sending an End-Of-Interrupt, it can receive nested interrupts. To consume as little stack space as possible, the number of nested interrupts must be limited.
- The time the physical device driver runs with interrupts disabled impacts system performance.

Resource Management

The various types of resource management for physical device drivers are as follows:

- Request packet queue management
- Memory management
- Semaphore management
- Character queue management.

Request Packet Queue Management

The strategy routine can either queue a request packet, or process it immediately. Typically, only READ and WRITE requests need to be queued because the device is busy. Other types of requests can usually be handled immediately by the strategy routine.

A block device driver, such as the physical Disk device driver, can process queued requests in any order. For instance, the block device driver can choose to sort the requests to optimize device access time. A character device driver must always handle queued requests in the order it received them; otherwise, mixed output could result.

The request packet queue is a linked list that contains a linkage field which allows the packets to be chained together. The device driver can manage its work queue of request packets with the DevHlp services for request queue management.

To use the request queue management DevHlp services, the physical device driver must allocate a DWORD variable as a queue header, with one header for each queue. The DWORD variable must be initialized to zero to indicate an empty linked list or the end of the linked list.

The DevHlp services use the queue header to identify a specific linked list of request packets and set the header to the first request packet in the list. The linkage field in the request packet then chains the request packet to another request packet.

Memory Management

The physical device driver must manage addressability to data across task-time and interrupt-time operations. DevHlp services are provided to allow the physical device driver to be independent of the CPU mode whether at task time or interrupt time. Addressability is particularly critical at interrupt time because the context of the current process might not cover the address space containing the data buffer that the hardware interrupt handler needs to access in order to move data.

To prevent an application from passing an unauthorized address, the physical device driver can use the DevHlp service, `VerifyAccess`, to validate the application's authority to access the memory. Because physical device drivers execute at the operating system privilege level (Ring 0), they have access rights to segments at all privilege levels. However, a well-balanced device driver must not allow an application to force it into accessing segments that the application does not own. This check applies to addresses that an application passes within a generic IOCTL request; the OS/2 kernel validates addresses for READ and WRITE requests. If an application passes a bad address to the physical device driver, the physical device driver could *cause a system halt* if it does not verify the caller's access authority. Once an address has been verified, the physical device driver can proceed with the I/O request.

The DevHlp services, `Lock` and `Unlock`, are used to fix in place a segment, which prevents the segment from being removed or swapped while the physical device driver needs access to it. The physical device driver does not need to lock segments for the READ or WRITE request packets. However, segments referenced in the generic IOCTL request packet need to be locked by the physical device driver, if it intends to access those segments at interrupt time.

Once a segment has been locked, the physical device driver can convert the virtual address into a physical address with the DevHlp, `VirtToPhys`, for later use at interrupt time.

The DevHlps, `AllocPhys` and `FreePhys`, allow the physical device driver to allocate and free a fixed amount of memory. The physical device driver must use the DevHlp, `PhysToVirt`, to obtain a virtual address to access the memory.

The physical device driver should choose to locate critical data structures or data transfer areas in its data segment. This is best for performance when access to the structures or buffers must take place at both task time and interrupt time.

Semaphore Management

There are two kinds of semaphores, RAM and system. These semaphores are characterized in the following table:

<i>Table 5-1. Semaphore Management</i>	
RAM Semaphore	System Semaphore
Created by allocating a DWORD variable	Created through a dynamic link function call
No OS/2 resource management is provided (frees the semaphore when the owner terminates)	Full OS/2 resource management provided (frees the semaphore and providing notification when the owner of the semaphore terminates)
Used to control operations among the physical device driver components.	Used to communicate to an application process.

RAM semaphores are defined by the semaphore user by allocating a DWORD variable and using the address in place of the handle in DevHlp semaphore services. The OS/2 operating system provides no resource management on RAM semaphores, such as releasing the semaphore when the owner terminates.

System semaphores are created by an application through a dynamic link function call. The OS/2 operating system provides full resource management on system semaphores, including releasing the semaphore and notification when the owner of the semaphore terminates.

System semaphores are typically used by a physical device driver to communicate with an application process. A physical device driver cannot create a system semaphore, although it can use the system semaphore that the application process has created. The application process must pass the application's handle to the physical device driver in a generic IOCTL. The physical device driver then uses the DevHlp service SemHandle to obtain a semaphore handle that it can use. The physical device driver must indicate in the SemHandle call that the system semaphore is IN-USE by the physical device driver. When the physical device driver no longer needs to use the system semaphore to communicate with the application, it must call the DevHlp SemHandle and specify that the system semaphore is NOT-IN-USE.

Character Queue Management

Character queues are used by character device drivers to buffer data. The two most frequently used structures for character buffers are the FIFO and the CIRCULAR buffer.

A character device driver can use the DevHlp services to manage a simple circular buffer for characters. The DevHlp services operate on the following character queue header:

```
CharQueue STRUC
    Qsize  DW      ?    ; Size of buffer in bytes
    QchROUT DW      ?    ; Index to next char out
    Qcount DW      ?    ; Count of characters in buffer
    Qbase  DB      ?    ; Start of buffer
CharQueue ENDS
```

Prior to using the character queue DevHlp services, a physical device driver must allocate the queue header and initialize the Qsize field. The DevHlp, QueueInit, must be called before calling any of the other character queue DevHlps. The other fields in the queue header are managed by the character queue DevHlps and do not need to be examined or altered by the physical device driver.

A character device driver is not required to use the character queue DevHlp services. A character device driver can define its own character buffer management tailored to the requirements of its buffer structure.

Requesting OS/2 Services

Because many of the functions of an OS/2 device driver are related to system operations, OS/2 services are available through the DevHlp functions. Access to OS/2 services is obtained at initialization.

Some OS/2 services are limited by the context in which the physical device driver is running. See Chapter 17, "Device Helper (DevHlp) Services" on page 17-1 for a list of the DevHlp services, descriptions, and limitations.

Note: In general, DevHlp services might briefly disable processor interrupts to prevent interruptions, but preserve the entry state of the interrupt flag upon exit from the service.

Limiting the Number of Nested Interrupts

Physical device driver interrupt handlers must be written to prevent uncontrolled stack growth of the interrupt stack. The interrupt handler must consume as little stack space as possible and limit the depth of nested interrupts.

The system's interrupt stack is configurable, that is, device drivers can indicate their interrupt stack usage by calling the RegisterStackUsage device helper routine during initialization. This allows the physical device drivers that require large stack space to be installed without decreasing available low memory for all device drivers. When calling RegisterStackUsage, a physical device driver provides data describing the interrupt handler for a specified hardware interrupt level (IRQ). RegisterStackUsage, therefore, is called by a physical device driver for each IRQ that it services. In addition to the interrupt stack size, this data includes the maximum number of interrupt levels that the physical device driver expects to nest. The OS/2 operating system uses this data to detect when unbounded nesting occurs and prevents nested interrupts from *causing a system halt*.

Physical device drivers that do processing after sending an End-Of-Interrupt (EOI) to the interrupt controller (8259) and enabling interrupts (STI), can receive nested interrupts. To prevent using excessive interrupt stack space, the physical device driver should keep internal flags to limit the amount of interrupt nesting. The amount of interrupt nesting is limited by performing all post-EOI processing at the first-level interrupt. Nested interrupts should avoid and, if possible, eliminate post-EOI processing. In all cases, the physical device driver must bound the number of levels of nesting to as few as possible (preferably two), and must never permit the levels of nesting to be unbounded. Limiting the amount of interrupt nesting to two levels can be implemented as follows:

- When a nested interrupt is encountered, a flag (or count) is set so the post-EOI processing can be done again by the first level interrupt, if necessary.
- On a first level interrupt, interrupts are enabled before doing the EOI and remain enabled during the post-EOI processing. Interrupts *must* be disabled and remain disabled before clearing the interrupt-in-progress flag and returning to the Interrupt Manager.
- On a nested interrupt, interrupts *must* be disabled and remain disabled before doing the EOI and returning to the Interrupt Manager.

Note: If a physical device driver needs to support more than one level of nested interrupts, it still must limit the number of nested interrupts that it handles. A physical device driver should avoid this if at all possible.

As device drivers allow more levels of nesting for their interrupt handlers, the potential exists for the entire interrupt stack to be consumed. This applies to all device drivers, regardless of the interrupt rate of the device being supported. It might seem that a device driver for a slow device need not follow this convention. However, if the system contains another device that has a high interrupt rate or many devices with more moderate interrupt rates, the time between occurrence of hardware interrupt and dispatch to interrupt handler can become greater than the interrupt rate of the slow device, and excessive nesting can occur.

Device Monitor Support in Character Device Drivers

The OS/2 operating system provides a mechanism for applications to directly access and intercept data flowing through data streams belonging to some character device drivers. This mechanism allows applications to filter (remove, insert, or modify) data passing through a character device by registering one or more character device monitors with the physical device driver. The mechanism requires support by the character device driver, as well as by the application.

A *character device monitor* is an application or part of an application that uses OS/2 dynamic link function calls to interact with:

- A physical device driver to gain access to its data streams by calling DosMonOpen, DosMonReg, and DosMonClose.
- A data stream to intercept data passing through the device by calling DosMonRead and DosMonWrite.

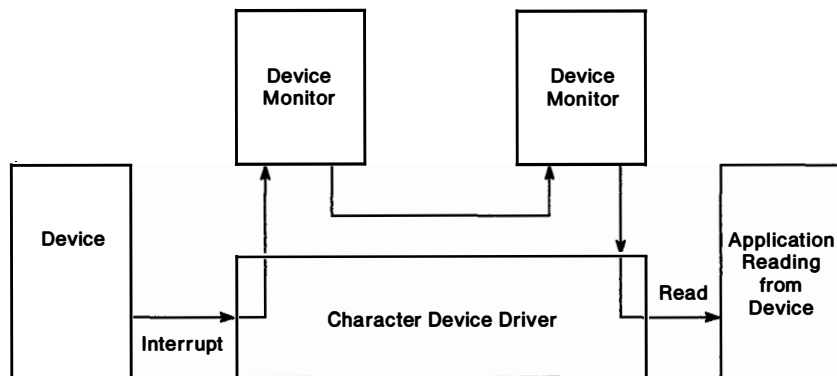


Figure 5-1. Input Device Monitor (for example: Mouse Monitor)

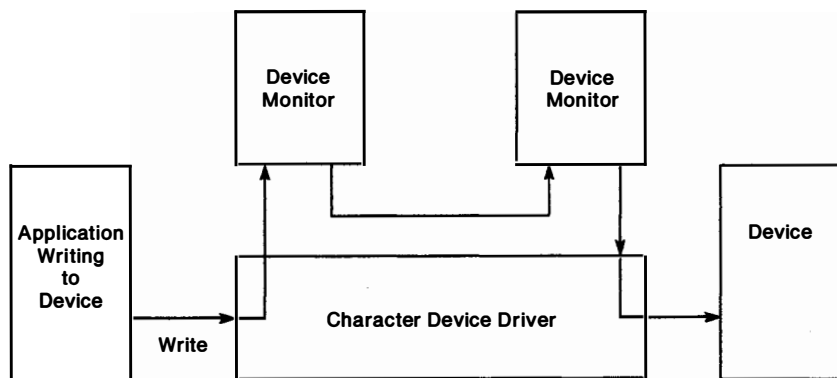


Figure 5-2. Output Device Monitor (for example: Parallel Port Monitor)

Character device monitors intercepting data from the same data stream belonging to a character device are linked together in a *monitor chain*. The first character device monitor in a monitor chain receives data directly from the physical device driver. This monitor filters the data and passes it on to the next character device monitor in the chain. This monitor then filters the filtered data and passes it on to the next character device monitor in the chain; and so forth. The last character device monitor in the chain passes the filtered data back to the physical device driver.

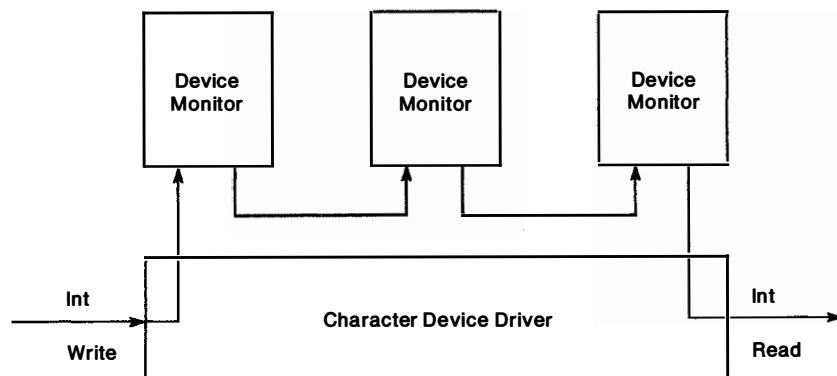


Figure 5-3. Device Monitor Chains

Character device drivers provide support for character device monitors by using the Monitor Dispatcher Device Helper services to:

1. Create monitor chains for their data streams by calling the MonitorCreate device helper routine
2. Register monitor buffers with their monitor chains on behalf of monitor applications by calling the Register device helper routine
3. Send data to their monitors by calling the MonWrite device helper routine
4. Flush all data from their monitors by calling the MonFlush device helper routine
5. Remove monitor buffers from their monitor chains on behalf of monitor applications by calling the DeRegister device helper routine.

Device Support

If device drivers are to be used on various system units, keep in mind that different system units can require different device drivers to run the same device. Notice that any device driver can be written to access hardware directly, even on a machine that has some form of BIOS interface. Writing directly to the hardware, however, makes the device driver device-dependent. Consequently, the physical device driver must be rewritten each time the hardware changes. Writing to a BIOS interface that hides device-dependent features allows a physical device driver to be supported across many devices.

Note: Refer to Appendix B, "Using Advanced BIOS" on page B-1 for information on ABIOS.

Consider how users of the device will make requests to the device. If users are expected to issue DosOpen, DosRead, DosWrite, and DosClose system calls, the physical device driver must support at least the READ and WRITE commands.

If users issue dynamic link calls to a subsystem, and the subsystem in turn issues DosOpen, DosDevIOCtl, and DosClose calls, the device driver must support the generic IOCtl commands.

To let users erase the contents of the physical device driver's I/O buffers, the driver must support the FLUSH INPUT and FLUSH OUTPUT commands. For information on strategy commands, their parameters, and service requirements, see Chapter 15, "Physical Device Driver Strategy Commands" on page 15-1.

There is more freedom to decide the commands character device drivers support than block device drivers written to run under the FAT-based file system. This does not mean that all block device driver must run under the FAT-based file system. The FAT-based file system is the user of the block device driver and expects certain functions to be supported.

Converting DOS Session Addresses Within IOCtl Requests

In OS/2 2.0, INT 21 IOCtl (AH = 44H) functions are passed to the OS/2 physical device driver in protect mode. Notice that the IOCtl packet can contain pointers. The addresses of the IOCtl packets are translated by the kernel since they are pointers. However, the kernel has no way to determine if there are pointers *inside* the packets. Consequently, the physical device driver is responsible for translation of any pointers imbedded in generic IOCtl packets.

When a physical device driver receives an IOCtl request, it must check the local information segment (see "GetDOSVar" on page 17-36) to determine if the calling process is executing in a DOS Session. The physical device driver does this by comparing the TypeProcess field to 1. If the value is equal to 1, then the physical device driver interprets any pointers within the Data or Parameter packets as real-mode segment:offset addresses.

The physical device driver must compute the linear address by shifting the segment value to the left by 4 bits, and adding the offset value. The physical device driver must allocate a GDT selector (see "AllocGDTSelector" on page 17-15), and convert the linear address to the GDT selector (see

“LinToGDTSelector” on page 17-41). The physical device driver now has a GDT selector pointing to the buffer passed in the IOCTL request. After verifying access to the buffer (see “VerifyAccess” on page 17-94) the physical device driver uses the selector to access the contents of the buffer. See the specific device IOCTL descriptions in Chapter 18, “Generic IOCTL Commands” for the categories and functions supported.

Inter-Device Driver Communication

A physical device driver can call and pass data to another physical device driver by using the Inter-device Driver Communication (IDC) mechanism provided by the OS/2 operating system. To communicate with each other, device drivers must maintain addressability, while being sensitive to interrupt-time performance.

A physical device driver uses its device driver header to indicate that another device driver can communicate with it. See the section “Physical Device Driver Header” on page 3-2 for a detailed description of the physical device driver header.

- The physical device driver must set bit 14 in the attribute field to indicate that it can participate in inter-device driver communication.
- The physical device driver must set up the offset to its entry point for other device drivers, which is identified as the IDC (inter-device driver communication) Entry Point.

Another device driver can communicate with this driver by calling its IDC Entry Point. The other device driver obtains the address of this entry point (and other information it needs to be able to call this driver) by calling the AttachDD device helper routine. This DevHlp service returns the address of the IDC Entry Point of the specified *named* device driver. See Chapter 17, “Device Helper (DevHlp) Services” on page 17-1 for more information.

The type and form of communication are defined by the physical device driver that provides the IDC Entry Point. Registers can be used to indicate the type of communication (for example, initialize a buffer, write data to a buffer), as well as the form of communication (for example, data structures). To communicate with a physical device driver, therefore, another device driver must use the published interface to the IDC Entry Point.

Chapter 6. Character Device Monitors

OS/2 2.0 provides a mechanism for applications to directly access and intercept data flowing through data streams belonging to some character device drivers. (See Figure 6-1.) This mechanism allows applications to filter (remove, insert, or modify) data passing through a character device by registering one or more character device monitors with the physical device driver. The mechanism requires support by the character device driver, as well as by the application.

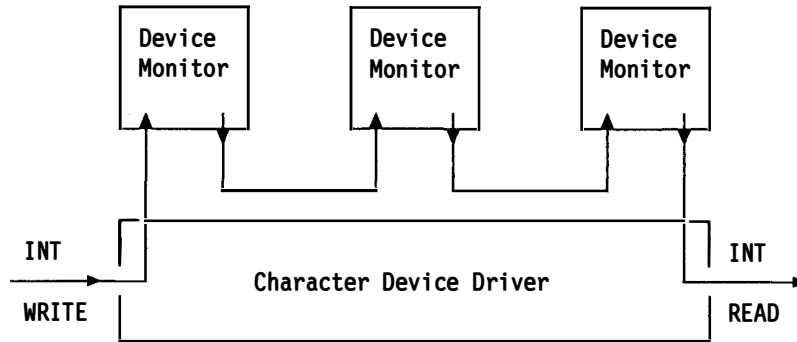


Figure 6-1. Data Interception by Device Monitors

Applications that monitor data passing through a character device can perform a variety of functions. They can serve as:

- User-extensions of a character device driver. Data passing through a data stream can be manipulated to modify the state of the device as viewed by another application.
- Dialog managers. Command sequences or control keys can be activated on a given keystroke. These include pop-up facilities and note pad applications.
- Language translators. Characters can be translated from one language to another.
- Redirection mechanisms. Data can be redirected from one device to another by character device monitors, as illustrated below. An application can register two monitors with two separate character devices. Data taken from the data stream belonging to the first device by the first monitor can be placed into the data stream belonging to the second device by the second monitor.

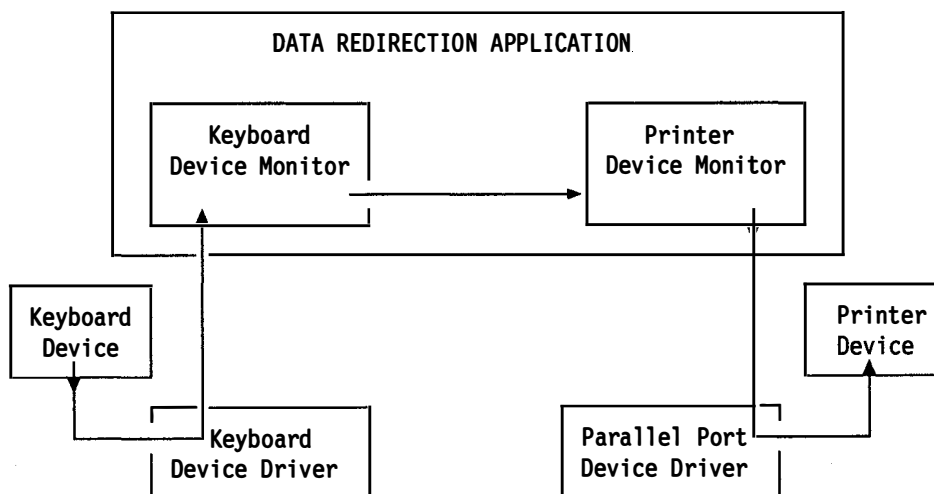


Figure 6-2. Data Redirection Using Device Monitors

OS/2 character device drivers that provide monitor support include the Keyboard, Mouse, and Parallel Port device drivers.

The OS/2 monitor mechanism includes the character device monitor process and the support required by a character device driver that enables device monitoring to be performed. Guidelines are given below for developing a character device monitor and for implementing monitor support in the character device driver.

Monitoring Character Device Data Streams

A physical device driver receives data from a device, or receives requests from applications to send data to a device. The physical Keyboard and Mouse device drivers receive data from the keyboard and mouse devices, respectively. The physical Parallel Port device driver receives requests from applications to write data to the printer device.

The flow of data between a physical device driver and its device is called a *data stream*. The physical device driver defines the data streams for its device, and associates the data it receives from a device or application with a particular data stream. Physical device drivers can have more than one data-stream model within, or between, sessions. For example, the physical Keyboard device driver supports three different data-stream models for DOS, Presentation Manager, and OS/2 sessions. Of these, only the OS/2-mode data stream supports keystroke monitors.

Note: OS/2 Version 1.2, Version 1.3, and Version 2.00 do not allow an application to register a keyboard or mouse device monitor for a Presentation Manager session.

There are several ways in which a physical device driver can define its data streams. When the physical Keyboard or Mouse device driver receives data from its device, it associates that data with the current foreground OS/2 session (which can be the Presentation Manager session) and places the data into the data stream defined for that session. These physical device drivers define data streams for each session. When the physical Parallel Port device driver receives a request from an application to write data to the printer, it places the data into the data stream defined for that printer device. This physical device driver defines data streams for each physical device.

A single data stream can be filtered by one or more character device monitors that are linked together in a monitor chain. That is, the first character device monitor in a monitor chain receives data directly from the physical device driver. This character device monitor filters the data and passes it on to the next character device monitor in the chain. This character device monitor filters the filtered data and passes it on to the next character device monitor in the chain, and so forth. The last character device monitor in the chain passes the filtered data back to the physical device driver.

Just as physical device drivers define their data streams, they also determine how character monitors are chained. The physical Keyboard and Mouse device drivers, for example, support a monitor chain for each OS/2 session. The physical Parallel Port device driver supports two monitor chains for each physical printer device. See Chapter 12, "Physical Mouse Device Driver," Chapter 11, "Physical Keyboard Device Driver," and Chapter 13, "Physical Parallel Port Device Driver" for a description of how these device drivers support device monitors.

Device Monitor Support Limitations

Data streams for character devices can only be monitored by a character-device monitor when the character device driver provides device monitor support. Applications, including both OS/2-mode and Presentation Manager applications, can monitor keystrokes and mouse clicks for all OS/2 sessions. However, the applications cannot monitor keystrokes and mouse clicks for Presentation Manager or DOS Sessions.

The keystroke and mouse click mechanism for Presentation Manager sessions differs from the mechanism for OS/2 sessions, as illustrated in Figure 6-3 on page 6-3. The physical Keyboard and Mouse device drivers manage keystrokes and mouse clicks, respectively, for each OS/2 session. These physical device drivers create monitor chains for each OS/2 session. Applications can register keystroke and mouse device monitors for each OS/2 session.

The Presentation Manager manages both keystrokes and mouse clicks for each of its extended sessions, but does not create monitor chains for each of its extended sessions. Applications cannot register a keystroke or mouse device monitor for a Presentation Manager extended session. Notice that applications cannot intercept keystrokes or mouse clicks associated with the Presentation Manager session through keystroke or mouse monitors.

However, all applications (including OS/2 and Presentation Manager applications) can monitor printer data. The physical Parallel Port device driver bases the definition of its data streams on the physical device. In general, data streams for character devices that do not base the definition of their data streams on sessions can be monitored by the character device monitor when the character device driver provides device monitor support.

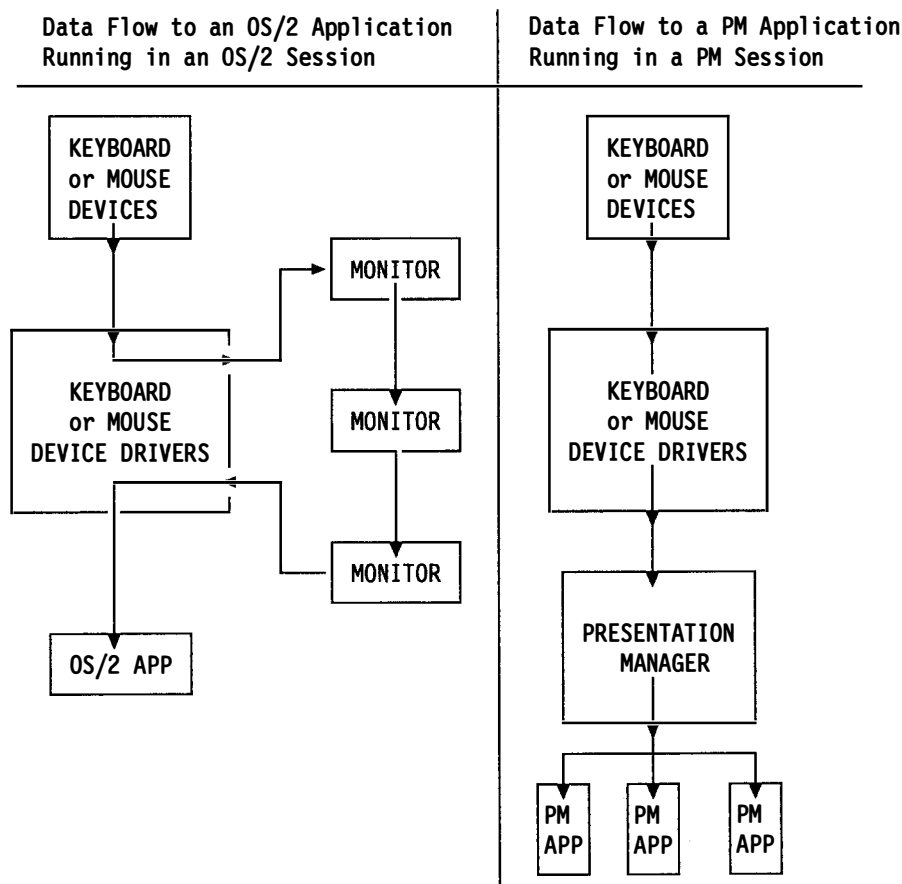


Figure 6-3. OS/2 Monitors and Presentation Manager Applications

A Presentation Manager application can filter keyboard and mouse events (messages) through *hook procedures*. These procedures are registered by an application and are called when certain events occur. More than one procedure can be called when a single event occurs. In this case, procedures are chained together so that each event is passed first to one procedure and then to the next, and so on down the chain. This chain of procedures is called a *hook chain*.

There are two kinds of hook procedures, system hook and queue hook. *System hook procedures* are called when an event occurs in the Presentation Manager's system queue. *Queue hook procedures* are called when an event occurs in the Presentation Manager application's queue. There are several kinds of events

that can be hooked. These include messages input to an application, and messages sent by an application. For detailed descriptions of hook procedures, refer to the *OS/2 2.0 Programming Guide Volume 2*, and the *OS/2 2.0 Presentation Manager Programming Reference*.

The OS/2 Monitor Mechanism

The OS/2 monitor mechanism consists of monitor buffers, monitor threads, and signals. This mechanism is supported by the OS/2 Monitor Dispatcher, which consists of:

- OS/2 monitor dispatcher functions, a set of five dynamic link routines:

- DosMonOpen
 - DosMonReg
 - DosMonClose
 - DosMonRead
 - DosMonWrite.

- Monitor Dispatcher Device Helper, a set of five OS/2 DevHlp services:

- MonitorCreate
 - Register
 - DeRegister
 - MonWrite
 - MonFlush.

The OS/2 monitor dispatcher functions are part of the MONCALLS dynamic link function library. These routines are loaded on demand at privilege level 2 and are callable from applications loaded at privilege level 2 or 3. Using these routines, applications can intercept and filter data passing through a device. The OS/2 monitor functions provide the interface for monitor applications to interact with the physical device driver (to request registration and termination of monitor activity) and the data stream.

The Monitor Dispatcher Device Helper (DevHlp) services are part of the OS/2 Device Helper, available to all character device drivers. Through these routines, a character device driver provides the support for applications that monitor its data streams. The Monitor Dispatcher Device Helper provides the interface for character device drivers to interact with the monitor dispatcher (for itself and for applications requesting registration and termination of monitors) and applications monitoring its data streams. In addition, the Monitor Dispatcher Device Helper provides the mechanism for passing data from one monitor to another.

Character device monitor applications are generally multi-threaded, with *child monitor threads* that take data from the data stream, and return filtered data to the data stream. Monitor threads use *signals* in two ways:

- When no data is available for the monitor in the data stream, a monitor thread can wait for a signal from the monitor dispatcher when data is available for the monitor. When the data stream cannot accept data from the monitor (for example, there is a blockage downstream in the monitor chain or at the physical device driver), a monitor thread will wait for a signal from the monitor dispatcher when the data stream can accept data from the monitor.
- When a monitor thread has finished taking data from, or returning data to, the data stream, it signals the monitor dispatcher that it has completed its work.

Notice that signaling between monitor threads is managed by the monitor dispatcher and is transparent to the monitor application.

Character Device Monitor Process

A character device monitor is an application, or part of an application, that uses standard OS/2 function calls to interact with the physical device driver and its data streams. A character device monitor can monitor only one data stream belonging to a character device driver. An application, however, can include one or more character device monitors that monitor one or more data streams belonging to one or more character device drivers. For example, an application can monitor keystrokes and mouse clicks for each OS/2 session. This application will include a character device monitor for each OS/2 session for each of the physical Keyboard and Mouse device drivers.

To monitor a data stream belonging to a character device driver, an application must first gain access to that data stream. An application gains access to a data stream by calling:

- **DosMonOpen** to establish a connection to the physical device driver. **DosMonOpen** returns a device handle for monitors to the application. The application will use this handle to direct subsequent **DosMonReg** and **DosMonClose** calls to the physical device driver.
- **DosMonReg** to register a monitor with a particular data stream belonging to the physical device driver. Once the monitor is installed in the monitor chain, the monitor dispatcher automatically moves data between monitors (if there are more than one in the chain) and returns filtered data to the physical device driver.

After an application has gained access to a data stream, it can remove, insert, modify, or view all characters passing through the data stream. An application intercepts data flowing through a data stream by calling:

- **DosMonRead** to take data from the data stream and place it into a private data area where the application can access it freely for filtering
- **DosMonWrite** to take filtered data from the application's private data area and return it to the data stream.

When an application no longer needs to monitor data streams belonging to a single character device driver, it must relinquish access to all data streams belonging to that physical device driver, in addition to terminating its monitor threads. An application terminates a monitor by calling:

- **DosMonClose** to close the device handle (that is, terminate the connection to the physical device driver for the monitors)
- **DosExit** to terminate its monitor threads that are directly accessing the data streams.

Refer to the *OS/2 1.3 Control Program Programming Reference* for more information on the 16-bit **Dosxxx** functions.

Figure 6-4 on page 6-6 illustrates pseudocode for a simple character device monitor.

A Simple, Single-Threaded Character Device Monitor

Get access to the device's data stream:

CALL DosMonOpen to open the device and get a monitor handle for the device
CALL DosSetPrt to set the priority of the monitor thread high
CALL DosMonReg to register a monitor for the data stream of the device

WHILE monitoring the data stream:

CALL DosMonRead to take a data record from the data stream
Filter the data record
CALL DosMonWrite to return the filtered data record to the data stream

END WHILE

After monitoring the data stream:

CALL DosMonClose to terminate the monitor and close the monitor handle to the device
CALL DosExit to terminate this application.

Figure 6-4. Pseudocode For a Simple Character Device Monitor

Character Device Driver with Monitor Support

For an application to monitor data passing through a character device, the character device driver must provide monitor support. For each data stream that can be monitored by applications, the character device driver must first create a monitor chain. For each monitor chain, the character device driver must define a *monitor chain buffer* in its first data segment, where the monitor dispatcher will place filtered data that has passed through all monitors registered with the monitor chain. In addition, the character device driver must define a *notification routine* within its first code segment. This routine is called by the monitor dispatcher when filtered data has been placed into the monitor chain buffer. Note that a character device driver creates a monitor chain by calling the DevHlp, MonitorCreate.

When an application registers a monitor (that is, calls DosMonReg), the character device driver receives the IOCTL request, "Function 40H — Register Monitor" from the monitor dispatcher on behalf of the application to register the monitor with the monitor chain belonging to one of its data streams. A character device driver registers a monitor with the monitor chain belonging to one of its data streams by calling the DevHlp, Register.

When one or more monitors are registered with a monitor chain belonging to one of its data streams, a character device driver can send data to its monitors. A character device driver sends data to monitors registered with one of its monitor chains by calling the MonWrite DevHlp service. Under certain conditions a character device driver can guarantee that all data sent to its monitors in a monitor chain has been filtered and returned to the data stream. A character device driver flushes all data from all monitors registered with a monitor chain by calling the DevHlp, MonFlush.

When a monitor application stops monitoring data streams belonging to a character device driver, or when a monitor application abnormally terminates (for example, the user presses the Ctrl-C key sequence), the character device driver receives a monitor close request from the file system. Because an application can have one or more monitors registered with one or more monitor chains belonging to the physical device driver, the character device driver must remove all monitors belonging to the application that are registered on any of its monitor chains. A character device driver removes monitors belonging to an application by calling the DevHlp service, DeRegister, for each monitor chain that belongs to each of its data streams.

Character Device Driver and Monitors

A character device driver and its monitors are interdependent. The implementation of the character device driver determines the ground rules for its monitors, including:

- The definition of its data streams and monitor chains
- The format of the data flowing through its monitor chains
- The rules on consuming, modifying, and returning data.

Because a monitor directly interacts with a data stream, there is a danger of severely impacting the data stream if a monitor is not well-behaved, that is, if the monitor does not adhere to the rules of the character device driver. The interdependence of a character device driver and its monitors can be directly demonstrated by presenting a description of the sequence of events occurring in the character device driver and its monitors during:

- Monitor registration and termination
- Movement of data through a monitor chain.

Registering and Terminating a Monitor: During monitor registration and termination, the monitor dispatcher communicates with the character device driver on behalf of the monitor application through the OS/2 file system and IOCTL interface. The application initiates OS/2 monitor functions, and the character device driver receives and responds to the corresponding requests. The following figure illustrates the monitor function calls made by the application, and the corresponding responses by the character device driver.

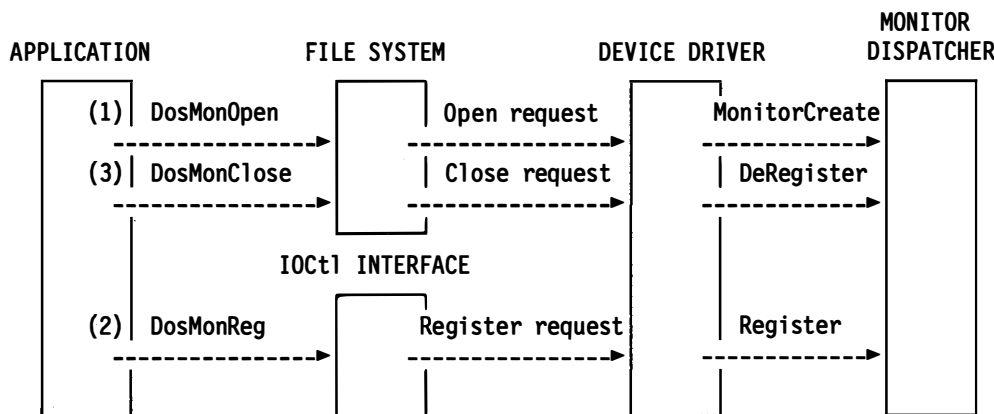


Figure 6-5. Application to the Device Driver Interface

For example, Character Device Driver Z defines only one data stream for its device. Application X wishes to monitor Character Device Driver Z's data stream:

1. Application X calls `DosMonOpen` to get a handle to Character Device Driver Z. Through the file system, the monitor dispatcher sends an Open request for monitors to Character Device Driver Z, on behalf of Application X.
 - If Character Device Driver Z has not already created a monitor chain for its data stream, Character Device Driver Z can call the `DevHlp, MonitorCreate`, to create the monitor chain.
2. Application X calls `DosMonReg` to register Monitor Y with the monitor chain for Character Device Driver Z's data stream. On behalf of Application X, the monitor dispatcher issues a monitor register IOCTL request to Character Device Driver Z.
 - If Character Device Driver Z has not already defined a monitor chain for its data stream, Character Device Driver Z must now call `MonitorCreate` to create the monitor chain
 - On receiving the monitor register IOCTL request, Character Device Driver Z calls the `DevHlp, Register`, so that the monitor dispatcher can install Application X in the monitor chain.

3. When Application X stops monitoring Character Device Driver Z's data stream, Application X calls `DosMonClose`.

- The monitor dispatcher sends a Close request for monitors through the file system to Character Device Driver Z on behalf of Application X
- On receiving the monitor close request, Character Device Driver Z must call the `DevHlp`, `DeRegister`, so that the monitor dispatcher can remove Application X from the monitor chain.

Data Passing Through a Monitor Chain: A character device can be an *input device* or an *output device*. A physical device driver receives data from an input device (for example, the keyboard or mouse) and makes it available to users through its API (Application Programming Interface) buffers. A physical device driver also receives requests from applications to send data to an output device. Using the example above, when Character Device Driver Z has created a monitor chain for its data stream, and Application X has registered Monitor Y with that monitor chain, data flows through the monitor chain as illustrated below:

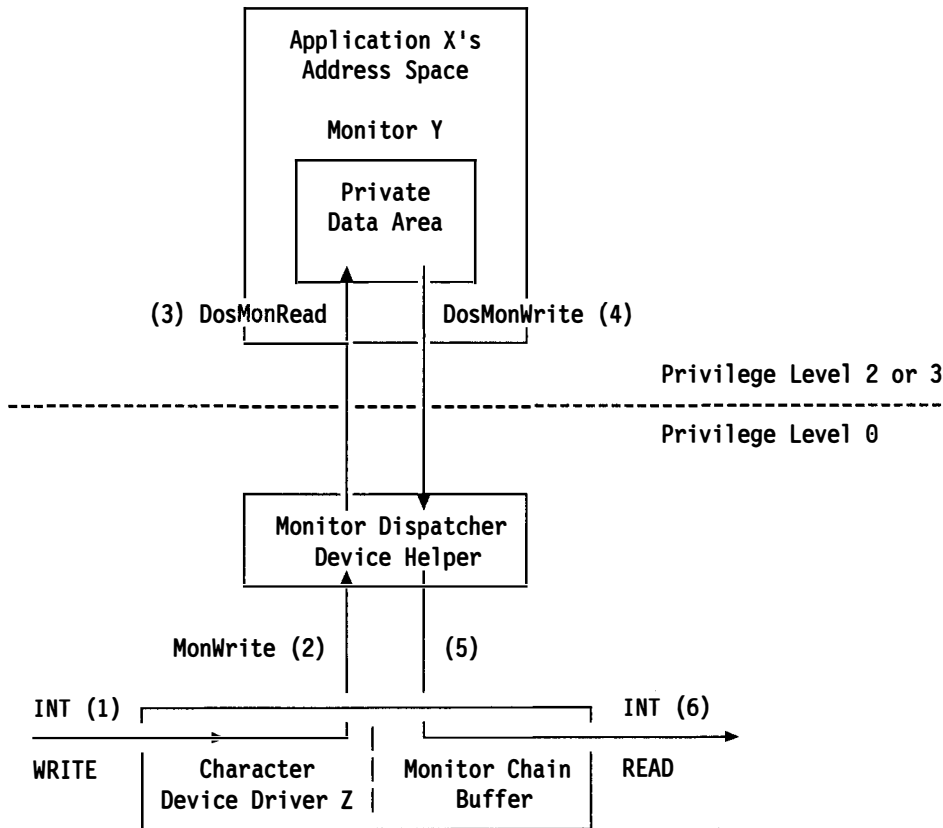


Figure 6-6. Data Flow Through a Monitor Chain

1. Character Device Driver Z receives data in one of the following ways:

- If Character Device Z is an input device, the data received by its physical device driver comes directly from the device
- If Character Device Z is an output device, the data received by its physical device driver comes from an application requesting output to the device.

2. Character Device Driver Z determines where to place the data, that is, into which data stream and monitor chain, if any, to place it. In this example, Character Device Driver Z has defined only one data stream for its device. Because a monitor chain has been created for the data stream, Character Device Driver Z places the data into the monitor chain by calling the `DevHlp` service, `MonWrite`.

3. Monitor Y calls `DosMonRead` to take data from the data stream, and place it into a private data area where it can freely access and process the data.
4. When Monitor Y has processed, or filtered, the data, it returns the filtered data to the data stream by calling `DosMonWrite`.
5. Since Monitor Y is the only monitor registered with the monitor chain, the monitor dispatcher automatically returns the filtered data from the output buffer of Monitor Y to the monitor chain buffer of Character Device Driver Z, and calls Character Device Driver Z's notification routine.
6. Character Device Driver Z processes the data received in its monitor chain buffer before returning to the monitor dispatcher:
 - If Character Device Z is an input device, Character Device Driver Z moves the filtered data from its monitor chain buffer into its API buffer, where it can be read from an application
 - If Character Device Z is an output device, character Device Driver Z sends the filtered data to the device.

OS/2 Monitor Functions

An application process that monitors data passing through a character device uses the OS/2 monitor functions to:

- Gain access to the data stream (`DosMonOpen`, `DosMonReg`)
- Take data from the data stream (`DosMonRead`), and return filtered data to the data stream (`DosMonWrite`)
- Relinquish access to the data stream (`DosMonClose`).

Monitor applications use the OS/2 monitor functions to communicate with a character device driver, and to directly intercept data from a data stream belonging to the character device driver. The OS/2 monitor functions are part of the OS/2 monitor dispatcher. They are shipped with OS/2 2.0 as the `MONCALLS` dynamic link subroutine library. These routines are loaded at privilege level 2, and are callable from applications running at privilege level 2 or 3. Each OS/2 monitor function uses the OS/2 System Trace Facility, tracing entry and exit parameters when tracing is enabled for monitors. This facility provides assistance for the monitor application developer during the debug phase of the application's development cycle.

OS/2 API standards require that a General Protection (GP) fault occurs when a null selector is passed as an address parameter to an API call. To conform to this standard, the monitor functions generate a GP fault when a null selector is passed as an address parameter. A pop-up message notifies the user that the process is being terminated.

DosMonOpen: An application that monitors data passing through a character device must first obtain a handle to the device by calling `DosMonOpen`. The application will need this handle to communicate with the character device driver in subsequent monitor functions, `DosMonReg` (for monitor registration) and `DosMonClose` (for monitor termination).

On calling `DosMonOpen`, the application pushes the parameters onto the stack, as shown in the following table. The address parameters are *selector:offset* addresses. Refer to the *OS/2 1.3 I/O Subsystems and Device Support Volume 1* for a detailed description of these parameters.

Parameter Description	Size
Address of Device Name String	DWORD
Address of Device Handle Returned	DWORD

DosMonOpen Parameter Stack

DosMonOpen calls DosOpen to open a handle to a character device for monitors. DosMonOpen returns the device handle returned from DosOpen to the application. The character device driver differentiates a DosOpen request from a DosMonOpen requests by the 08H value specified in the Status field of the request packet. On receiving the monitor open request, the character device driver can optionally issue a call to the DevHlp, MonitorCreate, to create a monitor chain. A character device driver can create a monitor chain anytime prior to issuing a call to the Register DevHlp, including at initialization time during device driver installation.

Note: An application needs to call DosMonOpen only once per character device. Repeated DosMonOpen calls by an application to open the same device for monitors will return the same handle. An application can register one or more monitors with data streams belonging to the same character device using the same handle. For example, an application can open the keyboard device for monitors (call DosMonOpen). Using the handle returned from this call, the application can register a monitor for each OS/2 session. Each OS/2 session has its own keystroke data stream and associated monitor chain.

DosMonReg: Before calling this function, an application previously must have called DosMonOpen to get a handle to the character device driver. The application needs this handle so that a monitor register request can be sent to the character device driver.

On calling DosMonReg, the application pushes the parameters onto the stack, as shown in the table below. The address parameters are selector:offset addresses. Refer to the *OS/2 1.3 I/O Subsystems and Device Support Volume 1* for a detailed description of the parameters to this function.

Parameter Description	Size
Device Handle Returned from DosMonOpen	WORD
Address of Monitor's Input Buffer	DWORD
Address of Monitor's Output Buffer	DWORD
Positional Placement Flag	WORD
Index (defined by the individual device driver)	WORD

DosMonReg Parameter Stack

The monitor's input and output buffers are allocated by the application from the same data segment. The first WORD of each buffer must contain the length of the private data area plus 20 bytes, length WORD inclusive. To maintain data movement through all monitor buffers in a monitor chain (more than one application can register monitors with the same monitor chain) the *minimum* size of these private data areas must be the length of the character device driver's monitor chain buffer. The monitor chain buffer is located at the end of a monitor chain, and receives filtered data that has passed through all monitors in the monitor chain. Because it is defined and owned by the character device driver, its length is documented in the descriptions of the individual character device drivers that support monitors. See the descriptions of the physical Keyboard, Mouse and Parallel Port device drivers for this information.

The Index parameter indicates the data stream for the character device that the application is monitoring. The individual character device driver defines this parameter. For example, index indicates the OS/2 session number for the physical Keyboard and Mouse device drivers. See the descriptions of the physical Keyboard, Mouse, and Parallel Port device drivers for details.

The Positional Placement Flag is used to specify the placement of a monitor's buffers within the monitor chain (first, last, or default). Location of a monitor within a monitor chain is relative to monitors that are already registered with the monitor chain. See "Positioning of Monitors in a Monitor Chain" on page 6-15 for details.

DosMonReg issues a monitor registration Category 10 "Function 40H — Register Monitor." On receiving a monitor register request, a physical device driver must call the Monitor Dispatcher Device Helper to:

1. Create a monitor chain by calling the MonitorCreate DevHlp routine, if no monitor chain was previously created
2. Establish the monitor placement within a monitor chain by calling the DevHlp service, Register.

Until there is a successful return from the call to DosMonReg, no character will enter the monitor's input buffer. That is, the monitor will not have access to the device driver's data stream. The application is responsible for synchronizing completion of the call to DosMonReg, and the subsequent monitoring of the data stream with device input into the data stream. For an example and solution to this problem, see the "Type-Ahead Characters" on page 6-22.

DosMonRead

After an application has registered as a monitor for a device, it can take data from the data stream for filtering by calling DosMonRead. DosMonRead has three primary actions:

1. Wait (optionally) for a signal from the monitor dispatcher that data is available to be read
2. Move that data from the data stream into a private data area where it can be accessed freely by the monitor for filtering
3. Signal the monitor dispatcher that data has been removed from the data stream.

On calling DosMonRead, the application pushes the parameters onto the stack as shown in the following table. The address parameters are selector:offset addresses of the following data areas within its address space:

- The input buffer address previously registered with the monitor chain for the device
- A private data area into which the data record read from the data stream is to be placed
- A Byte Count variable that on entry indicates the size of the private data area, and on exit indicates the size of the data record read from the data stream.

In addition, the application can specify whether to block (wait) if no data is available in the data stream.

Parameter Description	Size
Address of Monitor's Input Buffer	DWORD
Waitflag	WORD
Address of Private Data Area	DWORD
Address of Byte Count Variable	DWORD

DosMonRead Parameter Stack

For each DosMonRead call, a single data record is taken from the data stream and is placed into the monitor's private data area for filtering. A monitor data record is constructed by the physical device driver. It consists of a flag WORD, which can be followed by actual device data. Monitor data records are variable length (that is, some consist only of a flag WORD, while some consist of a flag WORD followed by actual device data of differing lengths). The maximum length of a monitor data record passing through a monitor chain is the length of the device driver's monitor chain buffer minus 2 bytes.

The physical device driver uses flags within the flag WORD to indicate the type of data that is part of this data record. The device driver also uses flags to indicate the action it expects its monitors to take when

this data record is received. In addition, physical device drivers have specific requirements for the return of data records to the data stream. Because of this, character device monitors should not indiscriminately consume data records from the data stream. See the chapters on the individual character device drivers for descriptions of their monitor records and expected actions.

Flushing a data stream is an important operation. At certain times it is necessary to guarantee that all data is removed from the data stream. At these times, a specially marked record called a *flush record* is placed into the data stream by the physical device driver and must pass through all monitors in the chain. The flush data record consists of a single flag WORD, with the third bit of the first byte set. It is the only data record created by the monitor dispatcher, and is placed into a monitor chain by the monitor dispatcher for the physical device driver. See "MonFlush" on page 17-45 for more information. When a flush record is received by a monitor, it must be returned to the data stream by calling DosMonWrite, after the monitor has taken the appropriate action. The action to be taken in response to the flush record varies with the type of device and type of support required. To guarantee that all data has been removed from all buffers in the monitor chain, the monitor dispatcher prevents the character device driver from placing additional data into the data stream until the flush record has passed through all monitors in the monitor chain.

Note: If the flush record is not returned to the data stream, the data stream will be severely and permanently impacted.

Separate monitor threads from the same application can call DosMonRead to take data from the data stream. To protect the data stream during data movement, the monitor dispatcher guarantees that only one thread at a time from a single process can take data from the data stream. Therefore, the application does not have to synchronize calls to DosMonRead made by its separate threads.

If a monitor thread calls DosMonRead during monitor termination (for example, DosMonClose has been called from another thread belonging to the same application), it will receive an error return indicating that there is no data present in the data stream. If a monitor thread is blocked on a call to DosMonRead when monitor termination begins, it will be awakened and receive the same error return. In both cases, the monitor thread must handle the error return code. The application is responsible for the orderly termination of its individual monitor threads.

DosMonWrite

After an application has registered as a monitor for a device, it can return filtered data to the data stream by calling DosMonWrite. DosMonWrite has three primary actions:

- Wait for a signal from the monitor dispatcher that data has been removed from the data stream if there is not enough room for the data
- Move filtered data from the monitor's private data area into the data stream
- Signal the monitor dispatcher that data has been placed into the data stream.

On calling DosMonWrite, the application pushes the parameters onto the stack as illustrated in the table below. The address parameters are selector:offset addresses of the following data areas within its address space:

- The output buffer address previously registered with the monitor chain for character device
- The private data area which contains the filtered data record to be returned to the data stream.

In addition, the application must specify a Byte Count variable, indicating the size of the data record to be returned to its output buffer.

Parameter Description	Size
Address of Monitor's Output Buffer	DWORD
Address of Private Data Area	DWORD
Byte Count	WORD

DosMonWrite Parameter Stack

For each call to `DosMonWrite`, a single filtered data record is placed into the data stream. See "DosMonRead" on page 6-11 for a description of monitor data record format, flushing monitor buffers, and restrictions on data record consumption by a monitor.

Separate monitor threads from the same application can call `DosMonWrite` to return filtered data to the same data stream. To protect the data stream during data movement, the monitor dispatcher guarantees that only one thread at a time from a single process can return data to a data stream. Therefore, the application does not have to synchronize calls to `DosMonWrite` made by its separate threads.

If a monitor thread calls `DosMonWrite` during monitor termination (for example, `DosMonClose` has been called from another thread belonging to the same application), it will receive an error return indicating that it cannot return data to the data stream. If a monitor thread is blocked on a call to `DosMonWrite` when monitor termination begins, it is awakened and receives the same error return. In both cases, the monitor thread must handle the error return code. The application is responsible for the orderly termination of its individual monitor threads.

DosMonClose

When an application no longer needs to monitor data passing through a character device, it must call `DosMonClose`, to remove all monitors that it has previously registered with the device. On calling `DosMonClose`, the application pushes the parameter onto the stack as illustrated below.

Parameter Description	Size
Device Handle	WORD

DosMonClose Parameter Stack

`DosMonClose` calls on the OS/2 File System to close the application's device handle for monitors by calling `DosClose`. The file system, in turn, sends a monitor close request to the physical device driver.

The character device driver differentiates a `DosClose` request from a `DosMonClose` request by the 08H value specified in the Status field of the request packet. At this time, the physical device driver must call on the monitor dispatcher to remove all monitors registered by the application by calling the `DevHlp`, `DeRegister`, for each of its monitor chains. (The application might have registered monitors with more than one monitor chain for the same device.) On the return from each call to `DeRegister`, if the monitor chain is empty (that is, there are no more monitors in the monitor chain) and the physical device driver no longer needs to use it, the device driver can (optionally) issue a call to the `DevHlp` service, `MonitorCreate`, with the Delete option to remove the monitor chain.

Note: The physical device driver must receive notification that a monitor process is terminating so that the monitor dispatcher can be called to remove the monitors from the monitor chain. The monitor dispatcher has the responsibility to guarantee an orderly removal of the monitor from the monitor chain with minimal data loss, while maintaining the integrity of data movement through other monitors in the monitor chain.

The OS/2 File System is already aware that a monitor has opened a handle to the device during a previous call to `DosMonOpen`. When a monitor process terminates (normally or abnormally) all

handles opened by that process are closed. The file system notifies the physical device driver that monitor termination is occurring by sending a monitor close request to the physical device driver.

If an application calls `DosExit` to terminate the process without calling `DosMonClose`, the file system sends a monitor close request to the physical device driver so that the monitor dispatcher can remove the monitors from all monitor chains.

Guidelines for a Character Device Monitor

The following is a summary of requirements and restrictions for character device monitors:

Monitor Buffers

Applications allocate their own monitor input and output buffers. A monitor input/output buffer pair must be allocated from the same segment. Before registering the monitor with a monitor chain belonging to a character device driver, the application is required to provide the size of its private data area (plus 20 bytes) in the first WORD of each buffer.

If `ERROR_MON_BUFFER_TOO_SMALL` is returned from the call to `DosMonReg`, the second WORD will contain the size of the character monitor buffer of the physical device driver.

The registered input and output buffers must be at least two WORDs (4 bytes) in length. These buffers can be discarded after the call to `DosMonReg`. The monitor dispatcher does not use the registered input and output buffers after monitor registration, but their addresses must be passed as parameters to the `DosMonRead` and `DosMonWrite` API functions. The address is used as a handle to determine which monitor is calling the API.

More than one application can monitor the same data stream. Monitors belonging to separate applications can have private buffers of various sizes. To guarantee data movement through a monitor chain, the monitor dispatcher defines a *minimum monitor buffer length*. This length is defined as the length of the character device driver's monitor chain buffer. The length specified in a monitor's input and output buffers must be at least this length plus 20 bytes. This is the recommended length of the private data area specified on a call to `DosMonRead`.

Because a monitor chain buffer is defined and owned by a character device driver, its length is documented in the descriptions of the individual device drivers that support monitors. See Chapter 11, "Physical Keyboard Device Driver," Chapter 12, "Physical Mouse Device Driver," and Chapter 13, "Physical Parallel Port Device Driver" for this information.

Character device drivers are deinstalled by a loadable device driver when specified with a `DEVICE=` statement in `CONFIG.SYS`. These loadable device drivers can implement the monitor buffer with a different size than the base physical device drivers. Character monitors must know the size of the character monitor buffer of the physical device driver in order to allocate their private data areas. The private data areas must be at least as large as the character monitor buffer of the physical device driver.

The character device monitor can issue `DosMonReg` first with a 4-byte buffer (the first two bytes containing the total length 4, and the second two bytes initialized to zero). On return from the call to `DosMonReg`, the second WORD of the 4-byte buffer will contain the size of the character monitor buffer of the physical device driver. The advantage with this approach is that the physical device driver can optimize the size of the character monitor buffer, and the character monitor can dynamically determine the buffer size. The character monitor then allocates the buffer size of the device driver, and re-issues the call to `DosMonReg`, replacing the first WORD of the input and output buffer with the length of the private data area plus 20 bytes.

Monitor Data Records

Only one monitor data record at a time can be taken from a data stream by calling `DosMonRead`, or returned to the data stream by calling `DosMonWrite`. A monitor data record is constructed by the physical device driver, and consists of a flag WORD, which can be followed by actual device data. Monitor data records are variable length. Some consist solely of a flag WORD, while some consist of a flag WORD followed by actual device data of differing lengths. To be consistent with the minimum buffer size defined for a monitor chain, the maximum length of a monitor data record passing through a monitor chain is the length of the device driver's monitor chain buffer minus 2 bytes. An application cannot return to the data stream a data record larger than this.

The physical device driver uses flags within the flag WORD to indicate the type of data that is part of this data record, and the action it expects its monitor to take on receiving this data record. Physical device drivers can have specific requirements for the return of data records to the data stream. Therefore, monitors should not indiscriminately consume data records from the data stream.

Flushing a data stream is an important operation. At certain times it is necessary to guarantee that all data is removed from the data stream. At these times, a specially marked record called a *flush record* is placed into the data stream by the physical device driver, and must pass through all monitors in the chain. The flush data record consists of a single flag WORD, with the third bit of the first byte set. It is the only data record created by the monitor dispatcher, and is placed into a monitor chain by the monitor dispatcher for the physical device driver. See "MonFlush" on page 17-45. When a flush record is received by a monitor, it must be returned to the data stream by calling `DosMonWrite`, after the monitor has taken the appropriate action. The action to be taken in response to the flush record varies with the type of device and type of support required.

To guarantee that all data has been removed from all monitors in the monitor chain, the monitor dispatcher prevents the physical device drive from placing additional data into the data stream until the flush record has passed through all monitors in the monitor chain.

Note: If the flush record is not returned to the data stream, the data stream will be severely and permanently impacted.

Positioning of Monitors in a Monitor Chain

Monitors are registered on a monitor chain with a positional preference parameter:

- 0 = DEFAULT
- 1 = FIRST
- 2 = LAST
- 3 = DEFAULT
- 4 = FIRST
- 5 = LAST

or monitor buffers are placed in a monitor chain in a position relative to monitors already registered with the monitor chain. The first monitor in a chain registered as *FIRST* will be at the head of the monitor chain. The next monitor registered as *FIRST* will follow the first monitor registered as *FIRST*, and so forth.

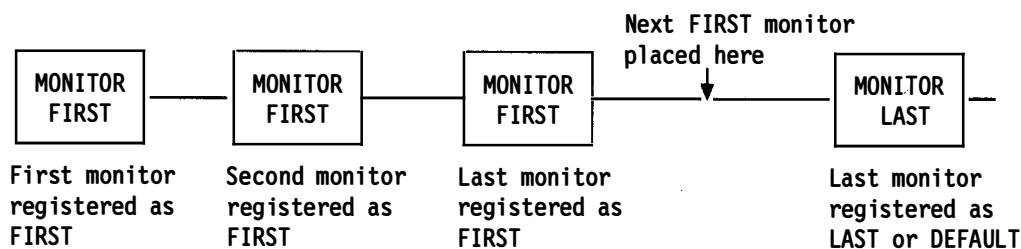


Figure 6-7. Monitors Registered as FIRST

Similarly, the first monitor registered as *LAST* will be at the very end of the monitor chain. The next monitor registered as *LAST* will precede the first monitor registered as *LAST*, and so forth.

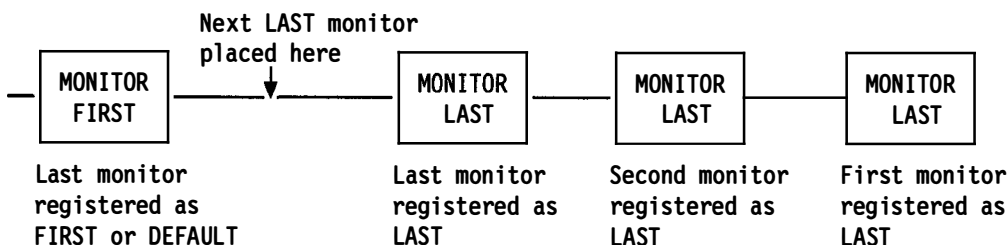


Figure 6-8. Monitors Registered as LAST

The first monitor registered as *DEFAULT* will precede the last monitor registered as *LAST*. The next monitor registered as *DEFAULT* will precede the first monitor registered as *DEFAULT*, and so forth.

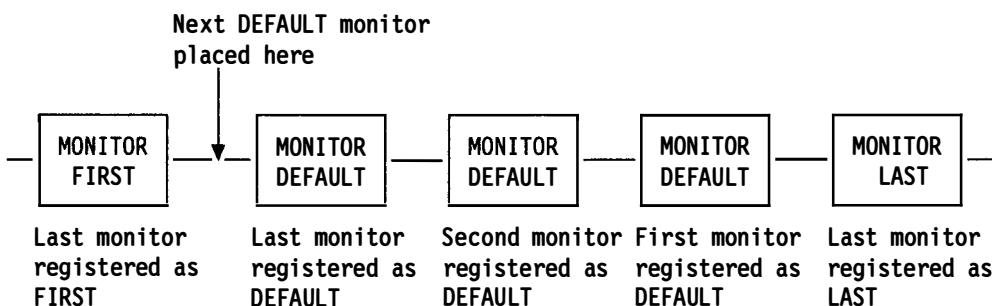


Figure 6-9. Monitors Registered in a DEFAULT Position

Note: It is possible for the last monitor registered as *LAST*, or the last monitor registered as *DEFAULT*, to be the first monitor in the chain.

Monitor Thread Priorities

To guarantee steady and reasonably fast data throughput in a monitor chain in this multi-threaded, multi-task environment, the priority of monitor threads must be set high. Because character device monitors are part of a data stream, they must process data records rapidly so that they do not delay I/O. A monitor application should be written so that the threads that actually read and write the monitor data run at a high priority. They should never perform operations such as I/O or semaphore waits that might delay them. The monitor application can have other threads running at normal priorities to handle such functions.

Threads responsible for moving keystroke data through a monitor chain must pay special attention to the thread priority. Keystroke monitor threads must execute within the time critical priority class. More specifically, these threads must execute at a priority level greater than or equal to the lowest level in the time critical priority class. The preferred level is Level 0. This applies to any threads that read (DosMonRead), process, or write (DosMonWrite) keystroke monitor data. In addition, the thread that makes the call to DosMonReg must also be executing in the time critical priority class.

It is generally recommended that a monitor application call DosCreateThread to create separate threads to intercept data from, and return filtered data to, a monitor chain. The application also must set the priority of these threads by calling DosSetPrt. See the *OS/2 1.3 Control Program Programming Reference* for descriptions of the DosCreateThread and DosSetPrt functions for more information concerning creating threads and priority classes, respectively.

Special Considerations for Character Device Monitors

The following section summarizes conditions to be considered when writing a character device monitor.

Performance

Several factors can affect the performance of a character device monitor, that is, its ability to intercept, filter and return data to a data stream quickly and efficiently. These include:

- Separate threads
- Task synchronization
- I/O requests
- Data consumption
- Expected responses
- Error handling.

Separate Threads: Because character device monitors are part of a data stream, they must process data records rapidly so that they do not delay I/O. A monitor application should be written so that the threads that actually read and write the monitor data run at a high priority. They should never perform operations such as I/O or semaphore waits that might delay them. The monitor application can have other threads running at normal priorities to handle such functions.

Task Synchronization: Semaphores can be used to synchronize and serialize the separate tasks of taking data from the data stream, filtering the data, and returning the filtered data to the data stream. For example, a monitor thread that has completed a call to DosMonRead can signal another monitor thread to filter the data. This second thread, having filtered the data, can signal a third monitor thread that the filtered data can be returned to the data stream by calling DosMonWrite. The third thread, having called DosMonWrite, can signal the first thread that it can take more data from the data stream.

Note: To avoid impacting the data stream severely or permanently, the application must take special care that its monitor threads do not wait on semaphores cleared by outside events, for example, waiting for I/O.

I/O Requests: Separate, non-monitor threads within a monitor application can make I/O requests of other devices which will not delay movement of data through the monitor. However, a monitor thread intercepting, filtering, and returning data to the data stream should not make I/O requests of the device whose data stream it is monitoring. *Such a request will result in a deadlock; data movement through the data stream will be permanently suspended.* For example, a thread belonging to a keystroke monitor application calls DosMonRead, filters the data, and calls DosMonWrite. Before calling DosMonWrite, however, the thread makes a KbdCharIn keyboard subsystem call to take data from the keyboard API buffer. Because no data has reached the physical Keyboard device driver's monitor chain buffer and,

therefore, the keyboard API buffer, the monitor thread blocks. The monitor thread cannot return the filtered data to the data stream, and therefore to the physical device driver's monitor chain buffer.

Data Consumption: To prevent other deadlocks, a character device monitor must follow the rules set by the character device driver and monitor dispatcher on data record consumption. The monitor dispatcher requires flush records to be returned to the data stream. Physical device drivers can have specific requirements for the return of data records to the data stream. For example, a character device driver can place a data record into a monitor chain, and then block until that data record is returned from the monitors into its monitor chain buffer. If a monitor removes that data from the data stream by calling `DosMonRead`, and does not return that data to the data stream by calling `DosMonWrite`, it will never reach the device driver's monitor chain buffer. The physical device driver will never unblock its blocked condition. Because of this, monitors should not indiscriminately consume data records from the data stream.

Expected Responses: The physical device driver uses flags with the flag `WORD` of a data record to indicate the type of data that is part of this data record, and the action the device driver expects its monitor to take when it receives this data record. The monitor can also be expected to indicate a response to the action requested by the physical device driver by changing the flags before returning the data record to the data stream. In this way, flags in a monitor data record are used in a dialog between the physical device driver and its monitors. Failure on the part of a monitor to respond as expected to a request from the physical device driver can result in a deadlock.

Error Handling: The developer of a character device monitor should not disregard error returns from monitor functions, and assume that *good data* is automatically received. A character device monitor must handle errors returned from all monitor function calls. *Failure to do so can result in severe and permanent impacting of the data stream.* For example, a monitor thread that ignores an error returned on a call to `DosMonRead` might attempt to `DosMonWrite` whatever data that is currently residing in the monitor's private data area. This data might be meaningful, that is, it might be the good data returned from the last call to `DosMonRead`. This data might not be meaningful, but might be the result of some other internal processing by the application. In either case, data unexpected by the physical device driver might be returned to the data stream.

Monitor Termination

An application is responsible for terminating its own monitor threads. The monitor dispatcher is responsible for removing the monitors from the monitor chains belonging to a character device driver. The removal is initiated by the character device driver when it receives a monitor close request. This occurs when:

- `DosMonClose` is called by the application
- `DosExit` (terminate all threads in a process) is called by the application
- The process is abnormally terminated by the user (for example, the Ctrl-C key sequence is pressed).

When the character device driver receives a monitor close request, it calls the `DeRegister Device Helper` routine for each of its monitor chains. Before returning to the physical device driver, the monitor dispatcher protects the integrity and order of data in the data stream, and guarantees minimal data loss, by:

- Locking out the application from subsequent calls to `DosMonRead` and `DosMonWrite`. The application is prevented from intercepting data from, and returning data to, the data stream.
- Draining all data in an orderly manner from the data stream.

The successful removal of monitors from a monitor chain is dependent on the ability of the physical device driver, or other monitors downstream in the monitor chain, to receive and process data from the data stream. If the monitor being removed from the monitor chain is not the last in the monitor chain, the next

monitor in the chain must not be blocked. Similarly, the physical device driver must not be blocked on a call to its notification routine; that is, it must be able to process data it receives in its monitor chain buffer.

During monitor termination, the monitor dispatcher prevents the application from touching the data stream by returning errors on calls to `DosMonRead` or `DosMonWrite`. In addition, errors are returned to monitor threads that are blocked on a call to `DosMonRead` or `DosMonWrite`. The application can then clean up the appropriate monitor threads. In both cases, the application is responsible for checking and handling the error returns.

During monitor termination there is always a risk of data loss. If the physical device driver or monitor dispatcher is blocked while waiting for critical data that is lost, *the system can be permanently impacted*. To reduce this risk, a monitor application should be *well-behaved*. See “Well-Behaved Monitor Applications.” Separate threads monitoring the data stream must be terminated before `DosMonClose` is called. This guarantees that all data taken from the data stream is returned to the data stream by the application. The application should stop calling `DosMonRead` and `DosMonWrite` before it calls `DosMonClose`. This can mean synchronizing the calls with semaphores.

The closing of monitor threads can also be synchronized by using signal handler or `exitlist` routines. *Signal handler* routines are called when the system sends a signal to the process to indicate that certain events are occurring, for example, the Ctrl-C key sequence has been pressed. *Exitlist* routines are a list of routines that are called when an application is terminating. In either case, these routines can be used to signal an application's separate monitor threads to prepare for termination. See the descriptions of the `DosSetSigHandler`, `DosHoldSignal`, and `DosExitList` functions in the *OS/2 1.3 Control Program Programming Reference* for details.

Well-Behaved Monitor Applications

A monitor application should be *well-behaved*, so that data loss is minimal, and system performance is not impacted. The following characteristics are typical of well-behaved monitor applications:

- Separate threads intercept data from the data stream (call `DosMonRead`), and return filtered data to the data stream (call `DosMonWrite`).
- Device monitor *rules* are followed, as defined by the monitor dispatcher and the individual character device driver whose data stream is being monitored. This includes definition of data streams, format of data records passing through the monitor chain, flags and expected actions, and consumption of data records.
- Error handling procedures exist for each function call.
- Semaphores and shared memory are used carefully so that blockages do not occur because of the monitor application, and thereby block the entire data stream.
- Data monitoring is terminated before calling `DosMonClose`, that is, all `DosMonRead` and `DosMonWrite` threads issue no more calls before `DosMonClose`.
- Signal handler or `exitlist` routines are included to coordinate the closing of monitor threads in an orderly manner.

A well-behaved, multi-threaded character device monitor is represented by the pseudocode in Figure 6-10 on page 6-20, Figure 6-11, and Figure 6-12 on page 6-21.

Application's Data

Define device-specific data:

Device name string (8 ASCII characters, with NULL at end)
Index (indicating data stream to be monitored)
Length of monitor chain buffer (minimum buffer size for all monitor buffers in monitor chain)
Monitor input buffer (4 bytes)
Monitor output buffer (4 bytes)

Define monitor data areas (the size of each of these areas is based on the size of the device driver's monitor chain buffer for the data stream monitored):

Private data area

Define additional monitor data:

Device handle (returned from DosMonOpen call)
Byte count variable (size of data record received by DosMonRead and returned by DosMonWrite)
Wait flag (wait for DosMonRead?)

Define a RAM semaphore to synchronize monitor threads:

Semaphore A

Define a variable to track error returns from DosMonRead and DosMonWrite:

SetError

Figure 6-10. *Data Definitions for a Well-Behaved Monitor Application*

An Application's Main Routine:

CALL DosMonOpen to open the device and get a monitor handle for the device
CALL DosSetPrtty to set the priority of the monitor thread high
CALL DosMonReg to register a monitor for the data stream of the device
CALL DosSemSet to set RAM Semaphore A, which is used to synchronize termination of the application
CALL DosCreateThread to create a separate thread that executes the application's monitor routine
CALL DosSemWait to wait for RAM Semaphore A to be cleared by the application's child thread
 executing the monitor routine
CALL DosMonClose to terminate the monitor and close the device handle
CALL DosExit to terminate this application

Figure 6-11. Main Routine of a Well-Behaved Monitor Application

An Application's Monitor Routine:

WHILE SetError = 0, the application's child thread will execute as long as there are no errors returned from calls to DosMonRead or DosMonWrite

CALL DosMonRead to take a data record from the data stream

SetError = error returned from DosMonRead

IF SetError = 0, THEN no error was returned from DosMonRead; continue.

 Filter the data.

 CALL DosMonWrite to return the filtered data to the data stream

 SetError = error return from DosMonWrite

ENDIF

END WHILE

CALL DosSemClear to clear RAM Semaphore A, to signal parent thread that monitoring of
 the data stream has been completed

CALL DosExit to terminate only this thread.

Figure 6-12. Monitor Routine of a Well-Behaved Application

Monitor Problems and Solutions

The checklist table shown below describes symptoms that can be experienced in a monitor environment, and suggests the problems that cause them.

Symptom	Problem
DosMonReg returns an invalid parameters error.	Possible trouble with the monitor buffers: <ul style="list-style-type: none"> • Are they in the same segment? • Does the first WORD of each buffer contain the length of the character monitor buffer of the physical device driver plus 20 bytes?
Monitor does not appear to be running. It is not receiving data from calls to DosMonRead, or returning data on calls to DosMonWrite.	Are the monitor threads running at a high priority?
Performance is poor. Data moves slowly through monitors.	Is the application doing complex processing in threads separate from the monitor (DosMonRead/DosMonWrite) threads?
Filtered data is not returned to the physical device driver.	Is the monitor consuming data records (that is, not returning them to the data stream by calling DosMonWrite)? If the device driver requires that these data records be returned, the monitor is blocking the data stream.
The system has stopped running.	Are the monitor or non-monitor threads polling the device's API buffer? The monitor threads might be unable to return data to the data stream. Therefore, the data will never reach the device driver's API buffer, and applications will continue to wait for data from READ commands.

Character Device Monitor Checklist

The following section describes special monitor problems and suggests solutions.

Type-Ahead Characters

An application can require monitoring all characters passing through a character device. In addition, the application can require that type-ahead characters are intercepted. *Type-ahead characters* are those keystrokes entered by a user after starting a program, and before prompts for information are given. Monitor applications must do some additional work to intercept these keystrokes.

There is a *time window* between the time when an application is invoked, and the time when the registration of the application's monitor is completed. During this time window, the application does not have access to the data stream. Because the monitor is not yet registered, it cannot intercept keystrokes. The physical device driver processes type-ahead keystrokes, placing them into its API buffer.

Note: If other applications have monitors registered with the monitor chain associated with the same data stream, type-ahead keystrokes will be filtered by these applications, and returned to the device driver for processing. Registration of other monitor applications with the same monitor chain is irrelevant.

To monitor type-ahead characters, an application must:

1. Register its monitor
2. Read and save type-ahead characters from the API buffer after monitor registration is completed, and before calling DosMonRead

3. Play back the saved type-ahead characters, and return them to the data stream by calling DosMonWrite
4. Monitor the data stream as usual, intercepting data from the data stream by calling DosMonRead, and returning filtered data to the data stream by calling DosMonWrite.

Pseudocode for the solution to this problem is illustrated below:

The Application

Description: A single-threaded application that requires monitoring all characters for a device, including type-ahead characters.

CALL DosMonOpen to open the device and get a monitor handle for the device

CALL DosSetPrtty to set the priority of the monitor thread high

CALL DosMonReg to register a monitor for the data stream of the device

Before data is intercepted, get and save any type-ahead data that the physical device driver has processed before DosMonReg returned.

WHILE the physical device driver's API buffer is not empty

 Get a character from the API buffer, using the appropriate device subsystem call

 Save the characters in a temporary buffer

END WHILE

Play back the type-ahead characters

WHILE the temporary buffer is not empty

 Get a character from the temporary buffer

 Build a monitor record, according to the specifications of the physical device driver

 Filter the data

 CALL DosMonWrite to return it to the data stream and, therefore, to the device driver. The physical device driver processes the data by placing it into its API buffer so that it can be received and processed by an application.

END WHILE

Monitor the data stream as usual.

WHILE monitoring the data stream

 CALL DosMonRead to take a data record from the data stream

 Filter the data record

 CALL DosMonWrite to return the filtered data record to the data stream

END WHILE

CALL DosMonClose to remove the monitor buffers from the monitor chain and close the device handle

CALL DosExit to terminate this application.

Figure 6-13. *Monitoring Type-Ahead Keystrokes*

Redirecting Data to Another Device

Character device monitors can be used to redirect data from one character device to another, as long the corresponding physical device drivers provide monitor support. This is done by:

1. Opening both devices
2. Registering a monitor with the first device
3. Registering a monitor with the second device
4. Intercepting data from the data stream belonging to the first device by calling `DosMonRead`
5. Using the data taken from the data stream belonging to the first device and creating a monitor record in the format defined by the second physical device driver
6. Placing the new data record into the data stream belonging to the second device by calling `DosMonWrite`.

The following pseudocode example shows how keystroke data can be redirected to the printer:

Redirecting Keystroke Data

```
CALL DosMonOpen to open keyboard device and obtain a monitor handle for the keyboard
CALL DosMonOpen to open printer device and obtain a monitor handle for the printer
CALL DosSetPrt to set priority of thread high
CALL DosMonReg to register a keyboard monitor for the data stream associated with the current session
CALL DosMonReg to register a printer monitor for the printer device's "data" monitor chain.
```

WHILE continuing to redirect data

```
CALL DosMonRead to take a data record from the keyboard data stream
CALL DosMonWrite to return the keyboard monitor data record to the keyboard device driver.
    Some data (for example, FLUSH records) must be returned to the keyboard data stream.
    Translate the keystroke monitor data record into a printer monitor data record.
CALL DosMonWrite to place the printer monitor data record into printer data stream.
```

END WHILE

```
CALL DosMonClose to terminate the printer monitor, and close the printer monitor handle
CALL DosMonClose to terminate the keyboard monitor, and close the keyboard monitor handle
CALL DosExit to terminate this application.
```

Figure 6-14. Redirecting Keystroke Data to the Printer

Note: Physical device drivers and the monitor dispatcher expect the return of certain monitor data records from its monitors. This requires that certain data not only be redirected to another data stream, but also be returned to its original data stream.

The advantage of this technique of redirecting data from one device to another is that it can be done without modifying or rewriting a character device driver. The disadvantage of using this technique is reduced performance. When performance is critical, data should be redirected between physical device drivers by using the inter-device driver communication mechanism provided by OS/2 2.0.

Providing Monitor Support in a Character Device Driver

Character device drivers provide support for character device monitors by using the Monitor Dispatcher Device Helper services to:

1. Create monitor chains for their data streams (MonitorCreate)
2. Register monitor with monitor chains (Register)
3. Send data to their monitors (MonWrite)
4. Flush all data from their monitors (MonFlush)
5. Remove monitors from monitor chains (DeRegister).

Character device drivers use the Monitor Dispatcher Device Helper services to manage the monitor chains associated with their data streams, and to move data between various monitors in their monitor chains. The Monitor Dispatcher Device Helper services are part of the OS/2 monitor dispatcher. All OS/2 Device Helper routines are part of the OS/2 kernel, and are loaded at the most privileged level (Ring 0). DevHlp services are invoked by passing parameters through registers, by loading a function code into the DL register and making a FAR call to the DevHlp interface routine. The AX register is used to return errors. The Carry flag is clear, if no errors are returned, and is set, if an error is returned. See Chapter 17, "Device Helper (DevHlp) Services" for descriptions of the individual Device Helper services.

Each OS/2 Monitor Dispatcher Device Helper service uses the OS/2 System Trace Facility. Entry and exit parameters are traced when tracing is enabled for monitors. This facility provides assistance for the character device driver developer during the debug phase of the development of a physical device driver. See the *OS/2 2.0 Control Program Programming Reference* for a description of the System Trace Facility.

MonitorCreate

As previously stated in the introduction to this chapter, individual physical device drivers determine how their data streams are defined. For example, the physical Keyboard and Mouse device drivers define data streams for each OS/2 session. The physical Parallel Port device drivers define data streams for each printer device. Each data stream can be monitored by a *chain* of one or more monitors. Before monitor registration can occur, however, the physical device driver must create a monitor chain for its data stream by calling the DevHlp, MonitorCreate.

MonitorCreate can be called at task time or at INIT time. The physical device driver can call MonitorCreate anytime prior to registering a monitor when:

- The physical device driver is installed. The physical device driver developer might prefer to create all monitor chains for each data stream during initialization of the physical device driver. Monitor chains are created, regardless of their usage. For example, a character device driver can create monitor chains for each OS/2 session during initialization. Because each monitor chain requires additional system resources, this can be a resource-expensive choice when monitors might not likely be registered on the monitor chains. The developer should consider this when designing and developing the device driver.
- The physical device driver receives a monitor open request.
- The physical device driver receives a monitor register request (when no monitor chain has been created, and before the DevHlp, Register, is called to register the monitor). The physical device driver developer might prefer to create all monitor chains as they are needed. For example, a data stream can be defined when an OS/2 session is started. The monitor chains associated with that data stream can be created only when an application calls DosMonReg to register a monitor.

On calling MonitorCreate, the physical device driver places the parameters to the call in the registers as illustrated in the table below. The parameters include the selector:offset addresses of the physical device driver's:

- **Monitor Chain Buffer.** This buffer is the last buffer in a monitor chain and must reside within the physical device driver's first data segment (that is, the device driver's header segment). The monitor dispatcher places filtered data (that is, data that has passed through all monitors in the monitor chain) into this buffer. The physical device driver processes this data by placing it into its API buffer, or by sending it to the device.

Notice that on calling `MonitorCreate`, the physical device driver must specify the length of the device driver's monitor chain buffer within the first WORD of the buffer, length WORD inclusive. Data records passing through a monitor chain will not exceed the length of the device driver's monitor chain buffer minus 2 bytes. The monitor dispatcher places filtered data into this buffer, starting at the second WORD of the buffer. The monitor dispatcher places the length of the data (in bytes) in the first WORD of the buffer, overwriting the original length WORD for the buffer.

- **Notification Routine.** This routine is called by the monitor dispatcher when filtered data is placed into the device driver's monitor chain buffer. This routine must reside in the device driver's first code segment.

In addition, the physical device driver specifies a monitor chain handle parameter. This parameter is used to indicate creation or deletion of a monitor chain. If the handle is zero when `MonitorCreate` is called, a new monitor chain is created and the *monitor chain handle* is returned to the physical device driver. If the handle is non-zero when `MonitorCreate` is called, the monitor chain with that handle is deleted.

Registers	Parameter Description
ES:SI	Address of Monitor Chain Buffer
DS:DI	Address of Notification Routine
AX	= 0 to create a new monitor chain (returns handle); < > 0 to delete a specific monitor chain.

Register Usage for `MonitorCreate`

The monitor chain handle is known only to the physical device driver and to the monitor dispatcher. The physical device driver uses this handle to identify the monitor chain to the monitor dispatcher on subsequent calls to the Monitor Dispatcher Device Helper services. For example, when the physical device driver registers a monitor for an application with its monitor chain, it uses the monitor chain handle returned from a previous call to `MonitorCreate` to indicate on which monitor chain the monitor should be registered.

The physical device driver maintains the list of monitor chain handles for its monitor chains, as well as the number of monitors registered on each monitor chain. When there are no monitors registered with a monitor chain, the monitor chain is *empty*. Notice that monitor chains are initially created *empty*. An empty monitor chain can be deleted by the physical device driver by calling `MonitorCreate` with the handle parameter set to the handle of the empty monitor chain. Physical device drivers that create their monitor chains as they are needed generally delete empty monitor chains when they are no longer needed.

Register

When an application calls `DosMonReg` to register a monitor with a character device driver, the device driver receives an IOCTL ("Function 40H – Register Monitor") monitor register request. Based on the `INDEX` parameter to `DosMonReg`, the device driver determines:

- Which data stream the application is requesting access to
- Which monitor chain the monitor should be registered on.

If no monitor chain has been defined or created for the data stream indicated, the physical device driver must first call `MonitorCreate` to create one. The physical device driver then calls `Register` to instruct the Monitor Dispatcher to add the monitors to the monitor chain.

The Register Device Helper routine can be called at task time. On calling Register, the physical device driver places the parameters to the call in the registers as illustrated in the following table. The parameters include:

- The selector:offset address of the monitor's input and output buffers
- A flag indicating positional placement in the monitor chain (see "Positioning of Monitors in a Monitor Chain" on page 6-15)
- The Process ID (PID) of the application requesting monitor registration
- The handle of the monitor chain on which it is being registered.

Registers	Parameter Description
ES	Selector of Segment Containing Monitor's Buffers
SI	Offset of Monitor's Input Buffer
DI	Offset of Monitor's Output Buffer
CX	Monitor's PID
DH	Positional Placement Indicator
AX	Monitor Chain Handle

Register Usage for Register

MonWrite

A character device driver places data into a monitor chain associated with one of its data streams by calling the DevHlp, MonWrite. If monitors are registered with the monitor chain (that is, the monitor chain is not empty), data placed into the monitor chain is automatically moved by the monitor dispatcher to the monitors in the chain. If monitors are not registered with the monitor chain (that is, the monitor chain is empty), data placed into the monitor chain is automatically moved by the monitor dispatcher into the device driver's monitor chain buffer. Notice that the physical device driver's notification routine is called when this occurs.

MonWrite can be called at task time or at interrupt time. On calling MonWrite, the physical device driver places the parameters to the call in the registers as illustrated in the following table. The parameters include:

- The selector:offset address of the data record to be placed into the monitor chain. This address must be located within the device driver's first data segment (that is, the device driver's header segment).
- The size of the data record (number of bytes).
- The handle of the monitor chain associated with the data stream.

In addition, the physical device driver can specify whether the monitor dispatcher should block or wait until it can place data into the monitor chain.

Registers	Parameter Description
DS:SI	Address of Data Record to Send to Monitors
CX	Size of Data Record (bytes)
AX	Device Driver's Handle for Monitor Chain
DH	Waitflag
DI:BX	Timeout Value, if DH = 2

Register Usage for MonWrite

A physical device driver can call MonWrite at interrupt time. If the physical device driver specifies the option to block (wait until data is successfully placed into the monitor chain) at interrupt time, the device

character device monitors

driver will receive an error from the monitor dispatcher if it is unable to place the data into the monitor chain. The physical device driver should re-issue the call so that no data is lost.

To prevent interruption while moving data during this call, the monitor dispatcher disables interrupts. This can be critical to the performance of a physical device driver that calls MonWrite to write data into an empty monitor chain. Under these circumstances, the monitor dispatcher:

1. Disables interrupts
2. Places the data into the device driver's monitor chain buffer
3. Calls the device driver's notification routine
4. Enables interrupts when the physical device driver returns from its notification routine.

When the physical device driver receives data in its monitor chain buffer, it must process that data quickly by placing it into its API buffer, or sending it to its device. The time between the call to the notification routine and the return to the monitor dispatcher must be minimal so that interrupts are not disabled too long.

MonFlush

A character device driver sometimes requires that all data placed into a monitor chain has passed through all monitors in the chain and has been returned to the physical device driver. A character device driver can call on the monitor dispatcher to *flush* all buffers in a monitor chain, that is, remove all data from all monitor buffers by calling the DevHlp service, MonFlush. This service can be called at task time. On calling MonFlush, the physical device driver places the parameters to the call in the registers as illustrated in the table below. The physical device driver specifies the handle of the monitor chain to be flushed.

Registers	Parameter Description
AX	Device Driver's Handle for Monitor Chain

Register Usage for MonFlush

The monitor dispatcher places a *flush record* (a single WORD data record with the third bit of the first byte set) into the monitor chain. To guarantee that all data has been removed from all buffers in the monitor chain, the monitor dispatcher prevents the physical device driver from placing additional data into the data stream until the flush record has passed through all monitors in the monitor chain.

Note: If the flush record is not returned to the data stream, the data stream will be severely and permanently impacted. Because placement of data into the monitor chain is suspended while a flush record passes through a monitor chain, *monitors must not consume flush records*. Flush records obtained from the data stream from a call to DosMonRead must be returned to the data stream on a call to DosMonWrite.

The action to be taken by a monitor application in response to the flush record varies with the type of device and type of support required.

DeRegister

When a monitor application terminates normally (by calling DosMonClose and DosMonExit) or abnormally (for example, Ctrl-C has been pressed or the monitor buffers have been corrupted), the file system sends a monitor close request to the character device driver. When the physical device driver receives this request, it must call on the monitor dispatcher to remove all monitors registered by the terminating process from all monitor chains belonging to the physical device driver. It calls the DeRegister Device Helper routine for each *non-empty chain*, that is, for each monitor chain for which there are monitors registered.

DeRegister is called at task time. On calling DeRegister, the physical device driver places the parameters to the call in the registers as illustrated in the following table. For each call to DeRegister, the physical

device driver provides the monitor dispatcher with the PID (Process ID) of the terminating application, and the handle of a non-empty monitor chain.

Registers	Parameter Description
AX	Device Driver's Handle for Monitor Chain
BX	PID of Monitor Process

Register Usage for DeRegister

For each call to DeRegister, the monitor dispatcher removes all monitors registered by the terminating process from the specified monitor chain, and returns to the physical device driver the number of monitors still registered with the chain. During the call to DeRegister, the monitor dispatcher reduces the risk for data loss by:

- Suspending subsequent calls to MonWrite and MonFlush to place data into the monitor chain until the monitors belonging to the terminating application have been removed from the monitor chain
- Marking the monitor as closed to subsequent data movement through DosMonReads and DosMonWrites
- Draining all data from the monitor in an orderly manner.

So that all data can be drained from the monitor, the physical device driver or another monitor downstream in the monitor chain must not be blocked. The physical device driver should not issue the call to DeRegister in either of the following situations:

- It has previously blocked on a monitor chain notification routine call because it cannot process the data received into its monitor chain buffer
- The return of a critical data record is pending (for example, a data record is placed into the monitor chain which the physical device driver is waiting to receive from the monitor chain before processing additional request packets).

Depending on timing and the well-behaved nature of the deregistering monitor, data records might be lost during monitor termination. A monitor that has not stopped taking data from, and returning data to, the data stream before terminating can inadvertently consume data records.

Guidelines for a Character Device Driver

For an application to monitor data passing through a character device, the character device driver must provide monitor support. A summary of buffer and code requirements for character device drivers follows.

Buffer Requirements

The character device driver must include a set of buffers in its data segments. It must include in its device header segment, that is, in its first data segment (the device driver can have multiple segments), a buffer in which to build the data record that is placed into a monitor chain and sent to its monitors on a call to MonWrite. For each data stream that can be monitored, the character device driver must also include in its first data a *monitor chain buffer* that receives filtered data from the monitor chain.

The first WORD of the monitor chain buffers must contain the length of the buffer, length WORD inclusive, each time MonitorCreate is called. The length of a monitor chain buffer defines:

- The *minimum* size of all buffers that are part of the monitor chain. The length specified in each input and output buffer belonging to all monitors that register with the monitor chain must be greater than, or equal to, the length of the device driver's monitor chain buffer plus 20 bytes.
- The *maximum* size of a data record placed into the chain of monitor buffers (that is, the length of the device driver's monitor chain buffer minus 2 bytes).

Code Requirements

The character device driver must include a set of routines and request handlers in the strategy routine in its code segment. Note that for each data stream that can be monitored, the character device driver must include in its strategy routine a notification routine that is called by the monitor dispatcher when it has placed a single filtered data record into the device driver's monitor chain buffer. The notification routine is called by the monitor dispatcher:

- When the monitor dispatcher automatically moves filtered data from the last monitor in a monitor chain into the device driver's monitor chain buffer
- When the physical device driver calls `MonWrite` to write data into an empty monitor chain.

When the notification routine is called, the device driver must process the data in its monitor chain buffer before returning to the monitor dispatcher. A character device driver must manage its monitor chains, creating them, deleting them, writing data into them, and flushing them. For each data stream that can be monitored by a chain of monitors, a device driver must define for the monitor dispatcher the location (addresses) of the notification routine and the monitor chain buffer by calling `MonitorCreate`.

The monitor dispatcher assigns a handle to the chain of monitors for the data stream. The physical device driver uses this handle when instructing the monitor dispatcher to perform device helper functions on the monitor chain (for example, `Register`, `MonWrite`, `MonFlush`, and `DeRegister`). The physical device driver can call `MonitorCreate` any time prior to issuing other monitor dispatcher device helper functions:

- At initialization time, that is, when the physical device driver is installed
- When it receives a monitor open request, if `MonitorCreate` has not been previously called to define the monitor chain
- When it receives a monitor register `IOctl` request, if `MonitorCreate` has not been previously called to define the monitor chain, and before the `DevHlp`, `Register`, is called to register a monitor with the monitor chain.

A physical device driver sends data to its monitors by placing it into the monitor chain. When monitors are registered on a monitor chain associated with a device driver's data stream, the device driver must place data received from the device or an application into the monitor chain so that it can be filtered. The physical device driver builds a data record in its data segment, and calls `MonWrite` to write that record into the monitor chain.

When a monitor chain associated with a device driver's data stream is empty (that is, a monitor chain has been defined during a call to `MonitorCreate`, but no monitors have been registered), the physical device driver can use its monitor support to place a data record directly into its monitor chain buffer by calling `MonWrite`. The monitor dispatcher automatically calls the device driver's notification routine to process the data.

A physical device driver can flush all data from the monitor chain associated with a data stream. When a device driver requires that all data be removed from the monitor chain, it issues a call to `MonFlush` to direct the monitor dispatcher to place a specially marked record called a *flush record* into the monitor chain. This record must pass through all monitors in the chain. All monitors in the chain that receive a flush record on a `DosMonRead` must return it to the monitor chain on a `DosMonWrite`. No new data records are placed into the monitor chain until the flush record reaches the device driver's monitor chain buffer.

A character device driver must handle monitor open, register, and close requests in its strategy routine. When an application calls `DosMonOpen` to get a handle to the device for monitors, a monitor open request is sent to the physical device driver. In response, a physical device driver can define a monitor chain for the data stream by calling `MonitorCreate`, if it has not previously done so.

When an application calls `DosMonReg` to register a monitor as part of the monitor chain for a specified data stream (see `INDEX` parameter) for a device, a monitor register request is sent to the physical device driver. In response, the device driver:

- Calls `MonitorCreate` to define the monitor chain, if it has not previously done so
- Uses the monitor chain handle returned from a previous call to `MonitorCreate`, and calls the `DevHlp`, `Register`, to instruct the monitor dispatcher to insert the monitor into the monitor chain for the specified data stream
- Tracks the number of monitors it has registered with each of its monitor chains.

When an application calls `DosMonClose` to terminate monitoring data passing to or from a device, a monitor close request is sent to the physical device driver. In response, the device driver:

- Must, for each monitor chain with which there are monitors currently registered, call the `DeRegister` Device Helper service to instruct the monitor dispatcher to remove all monitors associated with the process that issued the `DosMonClose` from the monitor chain.

Notice that a single process can register monitors for more than one data stream for a device. For example, a process can register a keystroke monitor for each OS/2 session. When this process terminates, the physical Keyboard device driver calls `DeRegister` for each monitor chain associated with an OS/2 session.

- On return from each call to `DeRegister` for a monitor chain, if there are no monitors registered in the monitor chain, calls `MonitorCreate` with the `Delete` option to delete the monitor chain.

Special Considerations for Character Device Drivers

This section summarizes conditions that should be considered when writing a character device driver with monitor support.

Device-Specific Monitor Information

The character device driver that provides device monitor support must carefully document the following information for its monitors:

- The definition of its data streams and their monitor chains. The monitor application developer requires this information for the `Index` parameter when calling `DosMonReg`.
- The length of its monitor chain buffers. The monitor application developer requires this information so that a large enough private data area can be allocated for the monitor.
- The description of its monitor record format, including a detailed description of the flag usage, and expected actions and restrictions of data record consumption. The monitor application developer requires this information to understand the data flowing through the data stream, and any actions or responses that the physical device driver expects.
- A description of conditions under which the physical device driver can block the data stream. The monitor application developer must ensure that the device monitor works with, and for, the physical device driver, not against it. The monitor application developer must understand expected blockages in the physical device driver (their causes, their effects) in order to handle them.

Performance

The performance of a character device driver with respect to its monitor chains can be affected by several factors:

- Performance can be degraded when interrupts are disabled too long. When a monitor chain is empty, a character device driver can place data directly into its monitor chain buffer by calling `MonWrite`. The monitor dispatcher disables interrupts, writes the data into the monitor chain buffer, calls the device driver's notification routine, and re-enables interrupts when the physical device driver returns to the monitor dispatcher. If the physical device driver fails to process the data placed into its monitor chain buffer quickly, interrupts can be disabled too long.

The monitor dispatcher must guarantee that the data placed into the device driver's monitor chain buffer during a call `MonWrite` is processed before new data can be placed into the buffer on subsequent calls. Because there is no shared semaphore or other means of signalling between the monitor dispatcher and the physical device driver to protect this data area from being overwritten, the monitor dispatcher protects the data in the buffer by disabling interrupts until the device driver has processed the data.

- Performance can be improved by increasing the size of the physical device driver's monitor chain buffer. Because the size of this buffer determines the size of all monitor buffers in a monitor chain, performance is improved during calls to `MonWrite` to place into the monitor chain, as well as during `DosMonRead` and `DosMonWrite` monitor function calls that intercept and return data to the data stream. As the size of the buffers increase, fewer blockages occur.

Note: When increasing a *published* size of a device driver's monitor chain buffer, there is a danger of preventing existing monitors from successful monitor registration since the size of the monitor's private data area is based on the size of the device driver's monitor chain buffer.

Device Driver Problems

When character device monitors are registered and active, a character device driver can severely and permanently impact its data stream or the entire system due to its:

- Failure to process data received in its monitor chain buffer quickly when the notification routine is called, especially when issuing `MonWrite` into an empty monitor chain. Blockages can occur in all monitor buffers in a monitor chain if the physical device driver is blocked too long.
- Inability to notify its monitors that there is a problem with the device. When a blockage occurs in a physical device driver because the device is disabled, the device driver can notify its monitors of the problem by placing a special monitor record into the monitor chain. The monitor can inform the user that there is a problem by displaying a message on the screen so that the user can re-enable the device.

However, blockages in all monitor buffers in the monitor chain due to a blockage at the physical device driver will prevent the device driver from notifying its monitors of a problem through a special monitor data record.

- Dependencies on the return of specific types of monitor data records from the monitors to the physical device driver. Documentation on individual device drivers must include requirements for returning special monitor data records to the monitor chain and to the physical device driver (for example, flush records). Even with well-behaved monitors, however, the physical device driver cannot guarantee that its monitors will return its critical data.

Note: The introduction of special monitor data records into a monitor chain monitored by applications previously written can *cause the system to halt*.

Sample Programs

DEMODB.ASM and MONITOR.ASM are sample MASM programs included in the *Developers Toolkit for OS/2 2.0 (Toolkit)* package. DEMODB.ASM is a sample character device driver for the DEMOD\$ device. MONITOR.ASM is a sample character device monitor for the DEMOD\$ device.

A Character Device Driver (DEMODB.ASM)

DEMODB.ASM demonstrates:

- How a device driver's strategy routine and timer handler are structured
- How queued requests are handled by the timer handler
- How a character device driver provides character device monitor support. When a device monitor is registered with the DEMOD\$ device, the physical device driver sends data to the monitor, and queues the filtered data returned from the monitor.

The user installs this physical device driver by including a `DEVICE =` statement in the CONFIG.SYS file. The user sends characters to the DEMOD\$ device, and the physical DEMODB device driver generates speaker tones with pitches corresponding to numeric characters.

The user writes data to the DEMOD\$ device by echoing (that is, directing) a string of characters to the DEMOD\$ device. The physical DEMODB device driver strategy routine receives these characters through the request packet interface. If a character device monitor (see MONITOR.ASM) is registered with the monitor chain associated with the DEMODB data stream, all characters are sent to the monitor by calling `MonWrite`. If no character device monitors are registered, the physical DEMODB device driver processes the original input characters.

The physical DEMODB device driver processes characters before queueing them on a character queue. The character string is scanned and only numerics are queued; all other characters are discarded. The timer handler wakes up every few milliseconds, and reads the character queue. The timer-interrupt generates speaker tones with a frequency that varies with the numeric character read from the queue.

A Character Device Monitor (MONITOR.ASM)

MONITOR.ASM demonstrates:

- Structuring a character device monitor
- Registering a monitor with a monitor chain
- Intercepting data from a data stream, filtering that data, and returning the filtered data to the data stream
- Handling special monitor records (the flush record)
- Terminating a monitor.

The user runs the Monitor application program as a detached process. The process registers a monitor with the physical DEMODB device driver, and intercepts all data sent to the DEMOD\$ device. The monitor filters the input data to the DEMOD\$ device by changing the characters A–J into 0–9 respectively. All other characters are unchanged. The monitor returns the filtered data to the data stream. The filtered data alters the function of the physical device driver by causing the physical device driver to generate speaker tones for the filtered characters A–J (received as numeric characters 0–9).

The monitor terminates itself whenever the character Q is intercepted from the data stream. After the monitor has been terminated, the characters A–J are no longer filtered. The physical DEMODB device driver generates no speaker tones for these non-numeric characters.

The RUNDEMOCMD batch file on the OS/2 sample programs diskette illustrates the function of the physical DEMODB device driver and how its monitor can alter that function:

character device monitors

1. Characters ABCDEFJ01256579 are sent to the DEMOD\$ device that is not monitored by a character device monitor. The physical DEMODB device driver receives seven non-numeric and eight numeric characters. It generates eight speaker tones corresponding to the numeric characters only.
2. A character device monitor (MONITOR.ASM) is registered with the physical DEMODB device driver.
3. Characters ABCDEFJ0125679 are sent to the DEMOD\$ device. The character device monitor changes the characters A – J to numeric characters 0 – 9. The physical DEMODB device driver receives seven filtered and eight non-filtered numeric characters. It generates fifteen speaker tones corresponding to the numeric characters.
4. Characters QABCDEFJ0125679 are sent to the DEMOD\$ device. When the character device monitor receives the Q, it terminates. The physical DEMODB device driver receives seven non-numeric and eight numeric characters. Speaker tones are generated for the eight numeric characters only.

Chapter 7. Installation of External Loadable Device Drivers

System installation provides the ability to install Device Support Diskettes. The user that has hardware not currently supported by OS/2 2.0 is allowed to install a specific hardware device driver from a Device Support Diskette during the system installation.

Note: The DDINSTAL program can be invoked by typing DDINSTAL at the command prompt, or at System Installation time.

The user can install external loadable device drivers through the system utility program DDINSTALL.EXE. A Device Driver Profile (a file with a DDP file name extension) must be provided (by the device driver programmer) to control the installation of the device driver using DDINSTALL.EXE. A diskette containing device drivers with their corresponding Device Driver Profiles is called a Device Support diskette. DDINSTALL.EXE installs all Device Driver Profiles on a Device Support diskette. When the installation is finished, DDINSTALL.EXE prompts the user to restart the system, causing the updated CONFIG.SYS file to take effect.

Note: DDINSTALL.EXE is not intended for the installation of presentation drivers.

DDINSTAL displays a list of the drivers found on the Device Support Diskette. This list is a multiple selection panel that lists the titles found in the :TITLE section of each Device Driver Profile (DDP). If no :TITLE section is found, the title defaults to "[No Title] -- filename.DDP." Each of the items selected are then installed.

Up to 24 DDP files are supported per diskette. Extra DDP files (over 24) are ignored. If there are 12 or less DDP files on the diskette, a non-scrolling multi-selection panel is used. Otherwise, a scrolling selection panel with a maximum of twenty-four selections is used. The user is then asked if there is another Device Support Diskette to be installed. If yes, this process is repeated for the next diskette.

If any files fail to copy because they replace open DLLs, or programs, then those files are copied into a temporary subdirectory. A DPP file is created with the names of the failing files, and the :CONFIG and :OS2INI statements. The user is then prompted to insert the Installation diskette and reboot. After the diskette boots, the DDINSTALL.EXE is called again to continue the device driver installation. The remaining files in the temporary subdirectory are copied, the :CONFIG statements are added to CONFIG.SYS, and the :OS2INI statements are added to OS2\OS2.INI. The temporary subdirectory is cleaned up, and the user is told that the device driver installation is complete and to reboot from the fixed disk.

If no files fail to copy from the device driver diskette, then the :CONFIG statements are added to CONFIG.SYS, and the OS2INI statements are added to OS2\OS2.INI. The user is then told that the device driver installation is complete and to reboot from the fixed disk.

The OS/2 operating system provides the functions necessary to install loadable device drivers to the OS/2 partition, and to update the system configuration files.

Device Driver Profile

The Device Driver Profile (DDP) is comprised of four sections, the :TITLE section, the :CONFIG section, the :FILES section, and the :OS2INI section. Each section is optional, and can appear any number of times, but the profile must contain at least one section. Blank lines are ignored and an asterisk (*) begins a comment, which is terminated by the end of the line.

:TITLE. This section contains a description of the DDP file. The first non-comment line in this section is displayed on the multi-selection panel as the title of the DDP file. The title should not exceed 64 characters in length. DDINSTAL is modified to read the :TITLE section for up to 24 DDP files on the diskette, and presents a multiple selection list (List Box) to the user. DDINSTAL then installs only the DDP files selected by the user.

installing device drivers

:CONFIG. The section describes the CONFIG.SYS statements required for loading the device driver when the system is initialized. Each line in this section is added to the end of CONFIG.SYS. These lines must be valid CONFIG.SYS statements. Because the complete line is appended to the CONFIG.SYS file, lines in this section should not be commented.

Before adding a line to CONFIG.SYS, DDINSTALL checks to see if the line already exists. This is done by performing a case-insensitive string comparison with the lines in CONFIG.SYS. If no match is found, the line is added. If a match is found, the line is ignored with no indication to the user.

:FILES. This section describes the files to be copied to the fixed disk by DDINSTALL.EXE. Each line in this section contains the source name and destination name of a file to be copied, separated by spaces. The default drive and path for the source name is **A:**, and the default drive and path for the destination name is the boot drive. The date, time and attributes of the file are preserved during the copy. If the destination path does not exist, it is created before the file is copied.

There are two methods of specifying the destination name. The first one is:

```
\devdrv\devdrv.sys
```

In this method, the path specified is appended to the driver letter entered on the command line. The second method is:

```
devdrv\devdrv.sys
```

In this method, the path specified is appended to the path entered on the command line.

Before copying a file to the destination, DDINSTALL checks to see if the file already exists. If it exists, its date/time stamp is compared to the date/time stamp of the source file. If the source file is more recent, it is copied to the destination. If the source file is older, it is not copied and no indication is given to the user. This ensures that the most recent files are installed on the destination.

:OS2INI. This section contains the OS2.INI statements. The format of OS2INI is:

```
VIO_xxx DEVICE=xxx.DLL [PtrDevP=ptr$]  
[PtrDevR=Ptr$]
```

This statement is subject to change, according to the Base Video Subsystem (BVS) specifications. The first parameter is **VIO_xxx**, where **xxx** is a descriptive name for the device. The second parameter must have a list of DLL and device names, separated by commas. The **PtrDevP** and **PtrDevR** tags are optional, and contain the pointer device driver names for real and protect modes. For example:

```
VIO_XGA DEVICE=XGA.DLL,MBUFFUP.DLL,XGA$ PtrDevP=XGAPTR$  
VIO_IBMCGA DEVICE=WINGA.DLL,BVSCGA.DLL PtrDevP=Pointer$
```

The following sample illustrates the format of a physical device driver profile:

```
***** A Sample Physical Device Driver Profile *****
* This is a physical device driver profile for a fictitious piece of hardware. It is used by *
* DDINSTALL to install loadable OS/2 device drivers and display drivers. It is not intended to *
* install Presentation Manager device drivers. This software consists of a physical device *
* driver, an I/O subsystem, and test programs. *
```

***** The Physical Device Driver *****

```
:TITLE                                * This section contains the title text.
A .DDP File Example
```

```
:CONFIG                                * This section contains CONFIG.SYS statements for the
DEVICE=\devdrv\devdrv.sys             * physical device driver.
```

```
:FILES                                * This section contains the files for the device driver
devdrv.sys\devdrv\devdrv.sys          * If the directory doesn't exist, it is created.
```

***** The I/O Subsystem *****

```
:CONFIG                                * This section contains CONFIG.SYS statements for the I/O
IOPL=devdrv                           * subsystem. The IOPL statement specifies the module name
                                         * in the DLL.
```

```
:FILES                                * This section contains files for the I/O subsystem.
devdrv.dll\os2\dll\devdrv.dll         * Copy to the DLL directory.
```

***** The Test Programs *****

```
:FILES                                * This section contains files for the test programs.
devdrv1.exe\devdrv\devdrv1.exe        * Test program 1.
devdrv2.exe\devdrv\devdrv2.exe        * Test program 2.
```

```
:OS2INI
video_displays vio_d10a display=devdrv1.dll
```



Part 3. OS/2 2.0 Physical Device Drivers

Chapter 8. Physical ASYNC (RS232-C) Communications Device Driver

The Asynchronous Communications (ASYNC) device driver enables OS/2 applications to utilize the Serial Communications (RS232-C) device hardware. The physical device driver allows an application program in the OS/2 mode to support full duplex communications while the device driver:

- Services the RS232-C port in an interrupt-driven manner
- Provides transmit and receive queues
- Provides different automatic control modes of the modem control signals
- Provides logical data stream flow control (XON/XOFF) for transmit and receive.

The user will normally want to use the physical ASYNC device driver either in conjunction with the spooler (for serial printers only), or with an application program that uses the RS232 enabling capabilities of the physical ASYNC device driver coupled with a serial device attached to the system.

The physical ASYNC device driver can be installed optionally by the user by using a `DEVICE=` command in `CONFIG.SYS`.

Hardware Support

The RS232-C ASYNC communications device driver supports any personal computer system based on an 80386-SX (or higher) microprocessor.

IBM PS/2 Micro Channel Adapter Support

The physical device driver supports a maximum of four ASYNC ports on a maximum of two different interrupt levels. The interrupt levels must have BIOS support, with one unit per Logical ID (LID) for the ASYNC Device ID. The only ASYNC devices supported on IBM PS/2, and the Extended Industry Standard Architecture (EISA) machines, are COM1, COM2, COM3, and COM4. These devices correspond to the first four LIDs in the BIOS common data area that have the architected ASYNC Device ID. These devices also correspond to the first four ASYNC addresses in the ROM BIOS 40: data area.

If a device has capabilities other than ASYNC, which cannot be utilized independently of the ASYNC capabilities (for example, as in the Advanced BIOS separate LID architecture), and if Advanced BIOS assigns the device the ASYNC Device ID, then that device can only be used for ASYNC in that power-on session.

If the device is assigned the ASYNC Device ID, and it has additional capabilities beyond supporting the RS232-C port (for example, a built-in modem), the physical device driver does not recognize those additional capabilities (and potential limitations). Also, the physical device driver does not inform any application program of those additional capabilities or limitations and does not limit the control of the RS232-C interface or the device to only those modes, which are acceptable to the extended hardware capabilities of that RS232-C port.

If the device is not assigned the ASYNC Device ID, it is not supported by this physical device driver. If an ASYNC device is not supported by the OS/2 operating system, but is recognized by Advanced BIOS as an ASYNC Device ID, the physical device driver can recognize and try to use that unsupported device, if it is COM1, COM2, COM3, or COM4.

AT Bus Adapter Support

The physical device driver for the IBM AT* bus machines supports two ASYN C ports, COM1 and COM2, each on separate levels. ASYN C ports with the following base I/O addresses are recognized by the physical device driver:

- 3F8H (must generate a level 4 interrupt)
- 2F8H (must generate a level 3 interrupt).

No other base I/O addresses are recognized or supported, and no other interrupt level combinations are supported. The physical ASYN C device driver for the AT* bus machine interfaces directly to the hardware and supports:

- IBM AT bus serial/parallel adapter based on the NS 16450 Universal Asynchronous Receiver Transmitter (UART) device
- Other compatible adapters based on the UART architecture (NS 16550, NS 16550A).

Attachment Support

The ASYN C physical device driver does not provide any support for devices attached to the RS232-C port. The physical device driver provides enabling support for the RS232-C interface itself. Application programs, subsystems, and systems programs provide the support needed to use devices attached to the RS232-C port.

The ability to support a device can be determined by understanding the level of RS232-C interface enabling support the physical device driver provides, along with the characteristics of the attachment hardware in question and the required functions to be supported.

The OS/2 operating system provides a mechanism where one or more additional drivers can be installed to support specific COM ports. This feature might be required for the following reasons:

- To allow an application program to support a special device not adequately supportable with this ASYN C device driver
- To allow additional COM ports (besides COM1-4 on IBM PS/2) to be supported
- To enhance the level of device driver function for a given COM port. (This might be required for certain subsystem support.)

RS232-C Interface

The ASYN C interface consists of separate read and transmit lines. There are two separate modem control signals whose output values can be controlled by the physical device driver:

- Data Terminal Ready (DTR)
- Request To Send (RTS).

There are four separate modem control signals whose input values are available to the physical device driver:

- Data Set Ready (DSR)
- Clear To Send (CTS)
- Data Carrier Detect (DCD), also known as Receive Line Signal Detect (RLSD)
- Ring Indicator (RI).

* Trademark of the IBM Corporation

The receive and transmit data lines have the following hardware characteristics:

- Logical 1 (Marking). More negative than -3 Volts. This state could mean no data.
- Logical 0 (Spacing). More positive than $+3$ Volts. This state could mean break condition.

The modem control signal lines have the following hardware characteristics:

- Function ON, when more positive than $+3$ Volts.
- Function OFF, when more negative than -3 Volts.

Hardware Support for Extended Hardware Buffering

This capability is a feature of the asynchronous communications port's serial controller device. Serial controllers which support this capability, such as the NS-16550A UART, are present in many IBM PS/2 systems and on a variety of IBM and non-IBM asynchronous communications adapters.

INS 8250, INS 8250-B Considerations

The following hardware defects cannot be compensated for in the physical device driver and can cause indeterminate function when used with the OS/2 physical ASYNC device driver:

Line Control Configurations: These devices are known to transmit bad data when configured for 5 data bits and 1.5 stop bits.

Receive Character Overrun Errors: These devices are known to occasionally drop received characters without posting the RECEIVE_OVERRUN error flag. Undetected data loss can result from this hardware deficiency. Application error-correction routines can be implemented to ensure accurate data transmission when these devices are being used.

Spurious Characters at Power-On: These devices can transmit a single random character at power-on. The connected device should not be expecting valid data to be received until after the physical device driver initialization routine has been run.

Supported Bit Rates on 16450 and Compatibles: The NS 16450 and other compatible UART devices (including the 8250- and 16550-Series UARTs) incorporate a Programmable Baud Generator feature that is driven as a function of the following constants:

CLOCK	= 1843200	; crystal frequency
CLOCK/16	= 115200	; after divider

Given these constants, the algorithm for determining which rates are supported is explained in the following examples:

- If 900 bps is specified, the bit rate is exactly 900 because it divides evenly into 115200 ($115200/900 = 128$). Bit rate, returned by IOCTL Function 61H, is 900.
- If 901 bps is specified, the bit rate does not change, and the IOCTL fails with an invalid parameter error because it cannot be supported within .01% ($115200/901 = 128$, $115200/128 = 900$, gives a .1111% error).
- If 907 bps is specified, the bit rate is 907.0866 because it can be supported within .01% ($115200/907 = 127$, $115200/127 = 907.0866$, gives a .0095% error). Bit rate, returned by Function 61H, is 907.
- If 110 bps is specified, the bit rate is 110.0287, even though the error is over .01% ($115200/110 = 1047$, $115200/1047 = 110.0287$, gives a .0260% error). Bit rate, returned by Function 61H, is 110.

Note: Where division is performed and the quotient is not a whole integer, an integer result is obtained by rounding off the fractional part of the quotient.

ASYNC (RS232-C) Device Driver Features

The device driver supports the ASYNC interface in an interrupt-driven manner. This allows the multi-tasking capabilities of the OS/2 operating system to be supported, while ASYNC data reception and transmission is taking place.

Warning: With any supported hardware, the physical device driver cannot absolutely guarantee accurate function, as there is a dependency on the hardware being driven. It is known, for example, that INS 8250 and INS 8250-B UART devices exhibit a number of deviations from their hardware specifications. In some cases, these deviations have been compensated for in the physical device driver design. Some of these deviations, however, cannot be resolved in software. The user should be familiar with the limitations and restrictions associated with such hardware.

With the current ASYNC hardware, when data is given to the transmit hardware, the data is transmitted at the physical RS232-C interface. When data is given to the transmit hardware, it has not yet been physically transmitted (at the RS232 interface). The data is considered completely transmitted by the transmit hardware at the physical RS232 interface when the transmit shift register of the UART is empty. The IOCTL, "Function 65H — Query Transmit Data Status" on page 18-43 can be used to determine this information.

The device driver *transmit* queue is a memory buffer between the operating system and the *transmit* hardware. The device driver *receive* queue is a memory buffer between the operating system and the receive hardware. Both are considered to be owned by the physical device driver because the physical device driver controls the data movement in and out of the transmit and receive queues. Algorithms for this data movement can change between releases of the physical device driver. Changes in the ASYNC hardware can cause changes in the data movement algorithms and external interfaces.

Data that applications send (made available by Write requests) are placed in the physical device driver transmit queue. When an interrupt occurs to tell the physical device driver that the hardware is ready for more data, the driver gives the transmit hardware more data from the transmit queue.

When an interrupt occurs to tell the physical device driver that the hardware has received data, that data is placed in the physical device driver receive queue. When the physical device driver gets a Read request (READ request packet) from the application, it fills the Read request from the receive queue.

At high bit rates, such as 19200 bits-per-second, a serial device supporting full-duplex asynchronous I/O can generate an interrupt every 260 microseconds (at 10 bits-per-character and one interrupt-per-character transmitted and received). This leads to excessive interrupt-time overhead in the multi-tasking, interrupt-driven, device driver.

To address this problem, serial devices with Extended Hardware Buffering capabilities (FIFO or First-In-First-Out buffers) have been developed. Many serially attached devices, however, which support the RS232-C interface, have been designed to operate with specific protocols that assume the system processes all data I/O one character at a time. The ASYNC physical device driver employs a software mechanism that automatically controls parameters to utilize the Extended Hardware Buffering capability, while compatibly supporting devices that use existing ASYNC device driver protocols.

The Automatic Protocol Override (system default) mode for Extended Hardware Buffering support partially utilizes only these performance advantages, while remaining fully compatible with the behavior of existing ASYNC device driver protocols (for example, Input Sensitivity using DSR). Applications and subsystems can disable certain device driver default settings in order to fully use the Extended Hardware Buffering capabilities. This results in a significant reduction of serial device interrupt processing overhead, and greatly increases the aggregate bit rates that can be supported across multiple active COM ports.

The size of the receive and transmit queues are available from the following IOCTLs:

- Query Number of Characters in Receive Queue. (Category 1 — Function 68H)

- Query Number of Characters in Transmit Queue. (Category 1 – Function 69H).

The physical device driver services each communications port independently. Requests issued to a given port have no effect on any other communications ports that the physical device driver might be servicing. The physical device driver processes READ and WRITE request packets independently for a given port. An application can be written to support simultaneous reception and transmission of data. In addition, the device driver can process an IOCTL request simultaneously with outstanding Read and Write requests.

The physical device driver does not schedule the processing of IOCTL requests. It processes the IOCTL request when received, regardless of what else it is doing. This can cause unexpected results if, for instance, the bit rate is modified while data reception or transmission is taking place. The application should issue only one IOCTL request at a time. If it issues another IOCTL request before the first IOCTL request completes, the results are UNDEFINED.

The device driver queues multiple READ and WRITE request packets independently and always begins processing the READ request packets in the order that they are received. It also begins processing the WRITE request packets in the order that they are received.

Note: The operating system does not guarantee that file system requests will be delivered to a device driver in the order in which they are issued by an application. This means that a request by one thread can get blocked in the operating system, thus allowing a subsequent request by a different thread for the same function (for example, DosWrite) to pass through and arrive ahead of the first thread at the physical device driver. This is true for synchronous operations performed by multiple threads, or asynchronous operations performed by the same thread.

Because of thread priority considerations and the system dynamics, the order observed by the application of completing requests of the same type might not be in the order that they were received by the device driver. The physical device driver always keeps the data in the same order in which the READ and WRITE request packets (of the same type) were received. There is no ordering or timing between different types of request packets.

The concept of a *First Level Open* is described in the section on “States of the ASYNC Device Driver” on page 8-8. A First Level Open occurs when the device driver receives an OPEN request packet for the port and the port is not already open (from a previous open without a matching close). A CLOSE request packet causing the physical device driver to process the next OPEN request packet as a First Level Open, is called a *Last Level Close*. Because the requests that an application issues sometimes get out of order before they reach the device driver, an application cannot consider a close a Last Level Close, until the CLOSE completes. If the application issues an Open request to the COM port, before a previously issued Close request is completed, then the results are UNDEFINED.

The Flush request can be completed before all the appropriate request packets (that have been queued by the device driver) have been flushed. The appropriate request packets eventually are flushed and return to the caller, based on their priority and the system dynamics. Once the Flush request has been processed, the appropriate request packets do not cause data to be transmitted (or received data to be moved) incorrectly.

The device driver supports different timeout processing characteristics and timeout settings for the Read and Write requests. Only the physical device driver is informed of when a given character is being transmitted or received at the hardware interface. Therefore, an application cannot expect to provide real-time flow control of data (in the middle of data transmission or reception) based on logical characters (XON/XOFF), or based on the state of the modem control signals by manually:

- Controlling or monitoring those modem control signals
- Monitoring the queue status
- Monitoring data moving across the link.

Alternatively, the physical device driver provides optional modes of operation to control the data flow through the RS232-C port automatically. OS/2 applications select which protocols are to be made active using IOCTls.

Output Modem Control Signals: In addition to allowing the application to control RTS and DTR directly, the physical device driver has different automatic control modes to control the value of the output modem control signals:

- Open and Close processing of DTR and RTS
- Disable/Enable DTR and RTS
- RTS toggling on transmit
- Input handshaking using DTR and RTS.

These control modes are described in the section on "States of the ASYNCR Device Driver" on page 8-8, and in the IOCTls description.

Note: The level of support provided by this device driver requires that DTR and RTS are turned on at least once, even if this puts the physical device driver in a mode where they will never be turned on again.

Input Modem Control Signals: Besides allowing the application to read directly the current state of DSR, CTS, DCD, and RI, the physical device driver has automatic modes that cause it to respond to the value that some input modem control signals can have:

- Output handshaking using CTS, DSR, DCD
- Input sensitivity using DSR.

These control modes are described in the section on "States of the ASYNCR Device Driver" on page 8-8 and in the IOCTls description. Additional information on the state of the input modem control signals is available by using the IOCTL "Function 72H - Query COM Event Information" on page 18-49.

Logical Flow Control (XON/XOFF): The application can attempt to manually control the flow of data by using the following IOCTls:

- Transmit Immediate. (Category 1 - Function 44H)
- Stop Transmit Behave as if XOFF Received. (Category 1 - Function 47H)
- Start Transmit Behave as if XON Received. (Category 1 - Function 48H).

The physical device driver automatically controls the flow of transmitted data based upon the reception of XON/XOFF characters. This is referred to as automatic transmit flow control (XON/XOFF). The physical device driver also attempts to control the flow of data that is received by automatically transmitting XON/XOFF characters to the system it is communicating with, based on the amount of space left in the receive queue. This is referred to as Automatic Receive Flow Control (XON/XOFF).

Support for Extended Hardware Buffering: Another significant feature of this device driver, is its exploitation of the Extended Hardware Buffering capabilities of the serial communications devices in many IBM systems and option adapters. *Extended Hardware Buffering* refers to the ability of the serial device servicing a COM port to buffer in hardware several characters, and release them all at one time on the occurrence of a single transmit or receive hardware interrupt. This capability significantly reduces the interrupt-driven I/O processing overhead required to service Transmit and Receive requests on a given COM port. On the devices that support the Extended Hardware Buffering capability, this significantly improves COM I/O throughput and improves data integrity for higher data transfer rates.

The Extended Hardware Buffering capabilities are automatically controlled under the default modes of the physical ASYNCR device driver. Automatic Protocol Override is a feature of the OS/2 ASYNCR device driver that automatically controls parameters relating to Extended Hardware Buffering. Systems and Adapters

that incorporate the *FIFO-mode* hardware feature in a manner fully compatible with the NS-16550A UART, are automatically enabled to run in Automatic Protocol Override mode.

Line Characteristics: IOCTLs can be used to control and read the bit rate, number of stop bits per character, number of data bits per character, and the parity characteristics of the line. See "States of the ASYNC Device Driver" on page 8-8.

Break and Error Processing: The device driver can be commanded to transmit a Break with an IOCTL (Category 1 – Functions 4BH, 45H). An application can detect where an error or break occurred in the input data stream by using Break Replacement Character Processing and Error Replacement Character Processing. This requires that certain binary byte combinations be reserved for this purpose.

State of the COM Port: The following IOCTLs can be used to determine the state of the COM port or if a given event happened. However, the exact timing relationship between this information and the specific data being received or transmitted at the time of the event is not available.

- Query COM Event Information (Category 1 – Function 72H)
- Query COM Status (Category 1 – Function 64H)
- Query COM Error (Category 1 – Function 6DH).

Event Notification: The device driver does not provide any capabilities of event notification. For example, the only way for an application to know that RI changed state or a Break condition occurred is to poll that status with the IOCTL, -- Heading 'C172H' unknown --. This should not be a problem for those applications that can use the automatic control modes of the physical device driver during the course of a communications dialog (for time-critical control functions). Polling could be adequate for non-time-critical event monitoring.

Error Alert Generation

The ASYNC physical device driver supports SNA Generic Alerts by generating Error Alerts, as defined under the OS/2 Logging Facility. Alerts are generated by the ASYNC driver whenever the OS/2 Logging Facility is enabled by the user at system initialization time.

Alerts may be generated only while the COM port is open and is processing a Write request (transmitting data). Write Timeout mode must be normal. (If Infinite Timeout mode is enabled, timeouts do not occur.) Error Alerts can be generated only when a Write Timeout occurs while waiting for:

- CTS to be asserted, when Transmit is disabled due to CTS being inactive. The Output Handshaking Using CTS mode must be enabled for this alert-generating condition to occur. This mode is enabled by default in the physical ASYNC device driver.
- DSR to be asserted, when Transmit is disabled due to DSR being inactive. The Output Handshaking Using DSR mode must be enabled for this alert-generating condition to occur. This mode is enabled by default in the physical ASYNC device driver.
- DCD to be asserted, when Transmit is disabled due to DCD being inactive. The Output Handshaking Using DCD mode must be enabled for this alert-generating condition to occur. This mode must be enabled by an application. It is not enabled by default in the ASYNC device driver.
- An XON to be received, when Transmit is disabled due to an XOFF being received. Automatic Transmit Flow Control mode must be enabled for this alert-generating condition to occur. This mode must be enabled by an application. It is not enabled by default in the ASYNC device driver.

Refer to the Set Device Control Block (DCB) Parameters Note on the IOCTL "Function 53H – Set DCB Parameters" on page 18-21.

States of the ASYNCR Device Driver

The different processing states of the physical ASYNCR device driver, the ASYNCR hardware, and the ASYNCR control signals are listed below:

- Automatic Receive Flow Control (XON/XOFF)
- Automatic Transmit Flow Control (XON/XOFF)
- Bit Rate
- Break Replacement Character
- Break Replacement Character Processing
- COM Event WORD and COM Error WORD
- Data Bits
- DTR and RTS
- DTR Control Mode
- Error Replacement Character
- Error Replacement Character Processing
- Extended Hardware Buffering.
- Input Sensitivity Using DSR
- Null Stripping
- Output Handshaking Using CTS, DSR, DCD
- Parity
- RTS Control Mode
- Read Timeout State
- Read Timeout Value
- Stop Bits
- Transmit Immediate
- Transmitting Break
- Write Timeout State
- Write Timeout Value
- XON/XOFF Characters

Each of the above states will be covered as follows:

- A brief description.
- The initial (default) value.
- The effect on the physical device driver when the physical device driver receives an OPEN request packet for the port and the port is not already open (from a previous OPEN without a matching CLOSE). This is called a First Level Open. If applicable, the way the state of the physical device driver is affected by a CLOSE request packet.
- How the MODE utility can be used to alter the state of this item or how the MODE utility will alter the state of this item.
- The effect on the physical device driver of the state of Extended Hardware Buffering. In particular, special considerations relating to Automatic Protocol Override are given where the protocol being described is affected by this feature of the physical ASYNCR device driver.

Automatic Receive Flow Control (XON/XOFF): When the physical device driver is enabled for this mode of operation, it transmits an XOFF, when its receive queue gets close to full, and an XON, when its receive queue is about half full.

The normal mode of Automatic Receive Flow Control causes the ASYNCR device driver to stop transmitting all data after it sends an XOFF character, until its receive queue lowers to about half full, when it sends the XON character. This behavior is suitable for most devices, but reduces transmit throughput significantly in Full-Duplex (Transmit and Receive) communications scenarios. By setting the Full-Duplex mode of Automatic Receive Flow Control, application data continues to be sent by the physical ASYNCR device driver after it sends the XOFF character. See IOCTL "Function 53H – Set DCB Parameters" on page 18-21.

<i>Initial Value</i>	Automatic Receive Flow Control is disabled.
<i>First Level Open</i>	No effect on whether the physical device driver is enabled or disabled for this mode of operation. The state of the physical device driver is reset to show that the last flow control character automatically transmitted was an XON, if it is enabled for this mode of operation.
<i>Close Considerations</i>	If the last automatically transmitted character by the physical device driver was an XOFF and a CLOSE request packet is received, (when, after processing this close request, the port is not open any more from another open without a close), the physical device driver automatically transmits an XON, if possible.
<i>Mode Utility</i>	Always disables Automatic Receive Flow Control.

Automatic Transmit Flow Control (XON/XOFF): When the physical device driver is enabled for this mode of operation, it stops sending data to the transmit hardware when an XOFF is received, and resumes sending data to the transmit hardware when an XON is received.

If this mode is enabled, Error Alerts can be generated when the OS/2 Logging Facility is enabled. If an external device sends an XOFF, but does not send an XON, transmit is blocked waiting for the XON to be received. If the XON is not received before the Write Timeout period expires, an Error Alert is generated.

<i>Initial Value</i>	Automatic transmit flow control is disabled.
<i>First Level Open</i>	No effect on whether the physical device driver is enabled or disabled for this mode of operation. The state of the physical device driver is reset to show that it has not received an XOFF, so it can transmit (due to automatic transmit flow control), if it is enabled for this mode of operation.
<i>Mode Utility</i>	User interface to enable/disable this mode of the physical device driver.
<i>Automatic Override</i>	When Automatic Transmit Flow Control is enabled, the physical device driver responds to receiving the XOFF, within a single character time. That is, Automatic Protocol Override controls the Extended Hardware Buffering capability so that only one character is buffered for transmit at a time, and the device generates an interrupt for every character received (Receive Trigger Level is set to 1).

Bit Rate: The bit rate determines the hardware setting for the data transfer rate, specified in bits per second, and is the speed for which the hardware is configured. See IOCTLs "Function 41H – Set Bit Rate" on page 18-8, and "Function 61H – Query Bit Rate" on page 18-39.

<i>Initial Value</i>	1200 bps
<i>First Level Open</i>	No effect
<i>Mode Utility</i>	User interface to change the bit rate.

Break Replacement Character: The device driver uses this character value, if Break Replacement Character Processing is enabled. See IOCTL "Function 53H – Set DCB Parameters" on page 18-21.

<i>Initial Value</i>	00H
<i>First Level Open</i>	Reset to 00H
<i>Mode Utility</i>	No effect.

Break Replacement Character Processing: If Break Replacement Character Processing is enabled, and the device driver detects a break condition, it places the break replacement character in the physical device driver receive queue. If Break Replacement Character Processing is disabled, the physical device driver does not place any character in the physical device driver receive queue, when it detects a break condition.

<i>Initial Value</i>	Break replacement character processing is disabled.
<i>First Level Open</i>	Break replacement character processing is disabled.
<i>Mode Utility</i>	No effect.

COM Event WORD and COM Error WORD: These two WORDs have bits, which show status of the COM port. When an event happens, the appropriate bits are turned on. The bits are cleared when the WORD is read with the appropriate IOCtrl. See IOCtrl "Function 72H – Query COM Event Information" on page 18-49 and "Function 6DH – Query COM Error" on page 18-48.

<i>Initial Value</i>	All defined bits are 0
<i>First Level Open</i>	All defined bits are 0
<i>Mode Utility</i>	Not applicable.

Data Bits: The number of bits that are contained in each character transmitted, or received by way of the communications hardware. See IOCtrl "Function 42H – Set Line Characteristics" on page 18-9 and "Function 62H – Query Line Characteristics" on page 18-40.

<i>Initial Value</i>	7 data bits
<i>First Level Open</i>	No effect
<i>Mode Utility</i>	User interface to change the number of data bits.

DTR and RTS: The value of the modem control signals Data Terminal Ready (DTR) and Request To Send (RTS) put out by the communications hardware. Each signal is controlled independently and can be either on or off. See IOCtrl "Function 46H – Set Modem Control Signals" on page 18-16 and "Function 66H – Query Modem Output Signals" on page 18-44.

<i>Initial Value</i>	When the physical device driver starts the port during device driver initialization, their values are turned off.
<i>First Level Open</i>	The signals are normally turned on, but there are many conditions that can cause the signals to be affected differently. See IOCtrl "Function 46H – Set Modem Control Signals" on page 18-16 and "Function 53H – Set DCB Parameters" on page 18-21. for a complete explanation.
<i>Close Considerations</i>	A CLOSE request packet (when, after processing this Close request, the port is no longer open from another OPEN without a CLOSE), causes DTR and RTS to be turned off, after the transmit hardware has completely transmitted all the data sent by the physical device driver. In addition, at least 10 additional character times must have elapsed (or one second, whichever is less).
<i>Mode Utility</i>	Not applicable for direct control. Indirect effects through altering processing modes of the physical device driver are possible.

DTR Control Mode: The control modes for DTR are:

- Enable
- Disable
- Input Handshaking.

The Enable and Disable control modes of DTR affect DTR processing during a First Level Open. When these control modes are set through the Category 1 – Function 53H IOCtrl, the value of the DTR signal can be modified immediately by the physical device driver. The action depends on the previous control mode of DTR, and the current value of the DTR modem control signal. If the control mode of DTR is Input Handshaking, then the device driver controls the DTR signal, depending on how full the receive queue is. The bits that control these states of the device driver are in the device control block.

<i>Initial Value</i>	Enable
<i>First Level Open</i>	No effect
<i>Mode Utility</i>	User interface to change the DTR Control Mode of the physical device driver.

Error Replacement Character: The character value that the physical device driver uses, if Error Replacement Character Processing is enabled.

<i>Initial Value</i>	00H
<i>First Level Open</i>	Reset to 00H
<i>Mode Utility</i>	No effect.

Error Replacement Character Processing: The processing that the physical device driver performs, when a received character had an error (parity, framing, overrun, or lack of receive queue space), is determined by whether Error Replacement Character Processing is enabled (active).

<i>Initial Value</i>	Error replacement character processing is disabled.
<i>First Level Open</i>	Error replacement character processing is disabled.
<i>Mode Utility</i>	No effect.

Extended Hardware Buffering: The extended hardware buffering (FIFO-mode) capabilities available in supported systems applies to the NS-16550A UART device and other fully compatible devices. These serial devices are installed on many IBM PS/2 system planars, and on various ASYNC communications adapter options. Refer to "Hardware Support for Extended Hardware Buffering" on page 8-3.

On those systems which incorporate serial devices that fully and compatibly support Extended Hardware Buffering, the OS/2 ASYNC device driver provides three modes for exploiting this feature:

- Enabled
- Disabled
- Automatic Protocol Override.

The default is to enable Automatic Protocol Override on that COM port. Automatic Protocol Override is an ASYNC device driver mode of operation that automatically controls various parameters of Extended Hardware Buffering, such as Receive Trigger Level and Transmit Buffer Load Count.

Automatic Protocol Override causes the Receive Trigger Level and Transmit Buffer Load Count to be adjusted according to the device driver handshaking protocols in effect. When Automatic Protocol Override mode is *on*, and the handshaking protocols are set to their default settings, the physical device driver partially exploits only the performance advantages of Extended Hardware Buffering. The default handshaking protocols are:

- Enabled for Input Sensitivity Using DSR
- Enable for Output Handshaking Using CTS and DSR
- Disable for Output Handshaking Using DCD
- Disabled for Automatic Transmit Flow Control.

If both Input Sensitivity Using DSR and Output Handshaking Using CTS and DSR are disabled, the Automatic Protocol Override causes the ASYNC device driver to automatically reset internal parameters (fully exploiting the Extended Hardware Buffering capabilities to the maximum extent possible).

The physical device driver's initialization default is to set Extended Hardware Buffering capabilities to the Automatic Protocol Override mode. An application or subsystem can alternatively set Extended Hardware Buffering to DISABLED, which causes the hardware to service transmit and receive interrupts one character at a time. It can also set Extended Hardware Buffering to ENABLED, which causes the physical device driver to set certain Extended Hardware Buffering parameters to specified levels, so that the serial device fully exploits its Extended Hardware Buffering capabilities to the maximum extent possible.

When Extended Hardware Buffering is set to ENABLED, the following serial device hardware capabilities are exploited for maximum performance benefit and increased receive data integrity:

- By setting the Transmit Buffer Load Count to 16, up to 16 characters are placed in the transmit hardware buffer (FIFO) during the processing of one *Transmit Holding Register Empty (THRE)* interrupt.

- A 16-character receive hardware buffer (FIFO) is available with the Receive Trigger Level set to 1, 4, 8, or 14 characters. The Receive Trigger Level determines when the receive hardware generates a *Receive Data Available* hardware interrupt.

If the physical device driver receives an Open request for a COM port that is not already open, it does not alter the Extended Hardware Buffering setting, which is in effect at that time. The ASYNCR physical device driver preserves this state across multiple Open and Close requests.

<i>Initial Value</i>	Automatic Protocol Override mode is on
<i>First Level Open</i>	No effect
<i>Mode Utility</i>	User interface to set the Extended Hardware Buffering mode to ENABLED, DISABLED, or to Automatic Protocol Override.

Input Sensitivity Using DSR: When the physical device driver is enabled for this mode of operation, and DSR is off, the physical device driver discards receive data.

<i>Initial Value</i>	Input Sensitivity using DSR is enabled
<i>First Level Open</i>	No effect
<i>Mode Utility</i>	User interface to enable/disable this mode of the physical device driver.
<i>Automatic Override</i>	When Input Sensitivity Using DSR is enabled, the physical device driver responds to the lowering of the DSR line within a single character time. That is, the Extended Hardware Buffering (Receive Trigger Level) is adjusted so that the device generates an interrupt for each character received. The full 16-character receive buffer is still available to help avoid any receive hardware overruns. The Transmit FIFO feature is also still enabled so that only one transmit interrupt occurs for every 16 characters transmitted.

Note: In situations where DSR can naturally drop from on to off at the end of a communications session, but is not normally toggled during the session, it is most advantageous to disable Input Sensitivity Using DSR, after the communications data transfer has begun. This achieves a significant performance benefit (under the control of Automatic Protocol Override), without the risk of losing valid data upon termination of the session. If, however, the DSR signal is expected to toggle frequently during a communications session, Input Sensitivity Using DSR should not be disabled, or *potential data loss can result*.

Null Stripping: If the physical device driver is enabled for null stripping, characters read in from the receive hardware (non-error or non-break) with a value of 00H are thrown away. These null characters are stripped (not checked for Automatic Transmit Flow Control), even if the XON or XOFF character has been set to 00H, and are not placed in the physical device driver receive queue.

<i>Initial Value</i>	Null stripping is disabled.
<i>First Level Open</i>	Null stripping is disabled.
<i>Mode Utility</i>	No effect.

Output Handshaking Using CTS, DSR, DCD: This mode of the physical device driver can be controlled independently for each modem control signal. When this mode is enabled, the physical device driver does not give data to the transmit hardware if the modem control signals are off (disabled). Data is not transmitted unless all the lines enabled for output handshaking are up.

If one of these modes is enabled, Error Alerts might be generated when the OS/2 Logging Facility is enabled. If an external device causes CTS, DCD, and/or DSR to become inactive, transmit is blocked waiting for the respective line to become active again. If the line does not become active before the Write Timeout period expires, an Error Alert is generated.

<i>Initial Value</i>	Output handshaking using CTS and DSR is enabled. Output handshaking using DCD is disabled.
<i>First Level Open</i>	No effect.
<i>Mode Utility</i>	User interface to enable/disable this mode of the physical device driver for CTS and DSR (independently). Mode always disables this mode of operation of the physical device driver for DCD.
<i>Automatic Override</i>	When Output Handshaking using Data Set Ready (DSR), Clear To Send (CTS), and/or Data Carrier Detect (DCD) is enabled, the physical device driver responds to the dropping modem line within a single character time. That is, Automatic Protocol Override controls the Extended Hardware Buffering capability so that only one character at a time is given to the transmit hardware (Transmit Buffer Load Count is set to 1). The Receive Trigger Level Level is unaffected.

Parity: Determines whether a parity bit exists and, if appropriate, what algorithm determines its value. See "Function 42H – Set Line Characteristics" on page 18-9 and "Function 62H – Query Line Characteristics" on page 18-40.

<i>Initial Value</i>	Even parity
<i>First Level Open</i>	No effect
<i>Mode Utility</i>	User interface to change the parity characteristics.

RTS Control Mode: The control modes for RTS are:

- Enable
- Disable
- Input Handshaking
- Toggling on Transmit.

The Enable and Disable control modes affect RTS processing during a First Level Open. When these control modes are set using the Category 1 – Function 53H IOCTL, the value of the RTS signal can be immediately modified by the physical device driver. The action depends on the previous control mode of RTS, and the current value of the RTS modem control signal. If the control mode of RTS is Input Handshaking, the device driver controls the RTS signal, depending on how full the receive queue is. If the control mode of RTS is Toggling on Transmit, then the physical device driver controls the RTS signal, depending on whether it is transmitting data. The bits that control these states of the physical device driver are in the device control block.

<i>Initial Value</i>	Enable
<i>First Level Open</i>	No effect
<i>Mode Utility</i>	User interface to change the RTS Control Mode of the physical device driver.

Read Timeout State: When the physical device driver processes a READ request packet, it can be with Normal, No-Wait, or Wait-For-Something Timeout processing. With *Normal* Timeout processing, if no data is received in the specified period of time, the request is completed. With *No-Wait* Timeout processing, the request obtains whatever data is available in the physical device driver receive queue (at the time the request is processed by the device driver) and returns. With *Wait-For-Something* Timeout processing, the request acts like No-Wait Timeout processing. However, if no data is available when the device driver processes the request, the physical device driver waits until some data is available, or until the request times out due to Normal Timeout processing.

<i>Initial Value</i>	Normal Read Timeout processing
<i>First Level Open</i>	The device driver is set to Normal Read Timeout processing
<i>Mode Utility</i>	No effect.

Read Timeout Value: The user specific value, in .01 seconds units (based on 0, where 0 = .01 seconds), is used for the Read Timeout processing, if *Normal Read Timeout* or *Wait For Something Timeout* processing is enabled.

<i>Initial Value</i>	1 minute
<i>First Level Open</i>	Set to 1 minute
<i>Mode Utility</i>	No effect.

Stop Bits: Determines the number of stop bits associated with each character transmitted, or received by way of the communications hardware.

<i>Initial Value</i>	1 stop bit
<i>First Level Open</i>	No effect
<i>Mode Utility</i>	User interface to change the number of stop bits.

Transmit Immediate: The device driver can be told to transmit a byte immediately, bypassing the normal file system Write requests (bypassing the data to be transmitted in the transmit queue). Only one character at a time can be waiting to be transmitted *immediately*.

<i>Initial Value</i>	There is no character waiting to be transmitted immediately.
<i>First Level Open</i>	There is no character waiting to be transmitted immediately.
<i>Close Considerations</i>	A CLOSE request packet, when after processing this close request the port is not open any more (from another open without a close), causes the device driver to attempt to transmit the character waiting to be transmitted immediately. If the physical device driver cannot transmit the character waiting to be transmitted immediately (see IOCTL Category 1 "Function 44H — Transmit Byte Immediate" on page 18-13), then it does not try to transmit the character and proceeds with the close processing.
<i>Mode Utility</i>	Not applicable.

Transmitting Break: The device driver can be transmitting a break. See IOCTLs "Function 4BH — Set Break ON" on page 18-20 and "Function 45H — Set Break OFF" on page 18-15.

<i>Initial Value</i>	Not transmitting a break.
<i>Close Considerations</i>	A CLOSE request packet, when after processing this close request the port is not open any more (from another open without a close), causes the device driver to tell the hardware not to transmit a break any more.
<i>Mode Utility</i>	Not applicable.

Write Timeout State: When the physical device driver processes a WRITE request packet, it can be with Normal or Infinite Timeout processing. With Normal Timeout processing, if no data is given to the transmit hardware within a specified amount of time, the request is completed. With Infinite Timeout processing, the request is completed only when all the data from the request has been given to the transmit hardware.

Error Alerts can be generated on the occurrence of Write Timeout, if the OS/2 Logging Facility is enabled. In order for an error alert to be generated by the physical ASYNCR device driver, the appropriate mode of operation must be enabled.

<i>Initial Value</i>	Normal Write Timeout processing
<i>First Level Open</i>	No effect on Write Timeout processing
<i>Mode Utility</i>	User interface to set Infinite or Normal Write Timeout processing.

Write Timeout Value: The user-specific value, in .01 seconds units (based on 0, where 0 = .01 seconds), is used for the Write Timeout processing, if Normal Write Timeout processing is enabled.

<i>Initial Value</i>	1 minute
<i>First Level Open</i>	Set to 1 minute
<i>Mode Utility</i>	No effect.

XON/XOFF Characters: The characters used for automatic transmit and automatic receive flow control.

<i>Initial Value</i>	XON is 11H, and XOFF is 13H.
<i>First Level Open</i>	The XON and XOFF characters are reset to their initial values.
<i>Mode Utility</i>	No effect.

Enhanced Mode

On COM ports with the Enhanced UART or compatibles, and the ABIOS function interface support, the physical ASYNC device driver runs I/O operations in enhanced mode by default, which is either DMA mode or enhanced FIFO mode. When the physical ASYNC device driver successfully allocates a DMA channel for the I/O operation, the operations are performed in DMA mode. Otherwise, the physical device driver defaults to enhanced FIFO-mode operation. In DMA mode, there are less CPU actions required for the data movements from port to memory, or memory to port, and the frequency of hardware interrupts can be significantly reduced. Using the Advanced BIOS function interfaces, DMA-mode operation maintains compatibility with existing ASYNC device driver protocols in most cases.

In enhanced FIFO mode, the full capacity of Extended Hardware Buffering (FIFO) is exploited to its maximum, automatically. Enhanced FIFO-mode operation does not require the Automatic Protocol Override mode to be compatible with the behavior of existing ASYNC device driver protocols. Automatic Protocol Override mode is ignored in enhanced FIFO-mode operation. In enhanced FIFO mode, the Advanced BIOS function interfaces are used to maintain full compatibility with existing communication protocols.

The current size of the receive queue is about 1KB, and the current size of the transmit queue is about 128 bytes. In DMA mode, there are two receive queues with the size of 1KB each. Since the current Status queue is used as the second receive queue, there is no memory size increase caused by this additional receive queue. The transmit queue size has been increased to 255 bytes in DMA mode for performance.

Note: These sizes are subject to change without notice and there should be no application dependency on their sizes being constant (even within the *boot cycle*).

DMA Channel Arbitration

A DMA channel has to be allocated to perform the DMA data transfer operation. With a limited number of DMA channels available in the system, contention over the channel is likely to occur between devices. The arbitration is done through the prioritized DMA arbitration level by the DMA chip. To fully operate in DMA mode, each port requires two different arbitration levels, one for receive and the other for transmit (for duplex mode operation). By default, the physical device driver operates the DMA channel in the following way for efficient use of the DMA resources, and to avoid sacrificing the performance of asynchronous I/O operations. The physical device driver also allows the user to control the DMA channel usage.

- Receive arbitration level is allocated per port at OPEN time, and deallocated at CLOSE time.
- Transmit arbitration level is allocated for each Write request, and deallocated when the request is done.

- If either of the arbitration level cannot be successfully allocated, then Transmit/Receive operations defaults to enhanced FIFO mode, in which case, the full FIFO capabilities are exploited without DMA. Also, in this case, the full capacity of the Enhanced RS232-C Enabling feature of the Enhanced UART is utilized.

The arbitration level can be set with the Hardware Reference Diskette.

DMA Receive Operation

The DMA capability allows higher bit-rate data transfer. This high-speed incoming *receive* data tends to fill the device driver receive queue rapidly. Depending on system overhead, if the rate of the arriving data to the receive queue is consistently higher than the rate of outgoing data from the receive queue, then eventually the receive queue will overflow. In reality, even with high bit rates, such as 345600 bit rate, the chance of a receive queue overrun is very unlikely since the data transfer rate from the receive queue to the user buffer outperforms the incoming data receive rate. For instance, at 345600 bps, it takes more than 7380 microseconds for the UART to receive 255 characters. It takes about 300 system clock cycles to move them from the receive queue to the user buffer, which is equivalent to 19 microseconds on a system with a clock frequency of 16 MHz.(00).

The device driver should set the Terminal Count, according to the length of the physical device driver receive queue, at the start of a receive operation. In DMA mode, the physical device driver receive queue is a flat queue, which is different from a conventional circular queue (where the head and tail pointers move in a circular fashion until the requested number of characters are reached). Using a flat queue necessitates the use of a second queue that can be switched to, when the first queue nears the end, in order to avoid buffer overrun.

The Enhanced UART has a Receive Character Count Register, which can be programmed to generate a hardware receive interrupt when RCCR reaches 0. Unlike the conventional PIO mode receive operation, in DMA mode, characters can still be transferred by the DMA chip, even while the hardware interrupt for the Receive Character Count Register = 0 is being serviced.

The Enhanced UART also provides a feature, which can be programmed to generate an interrupt, when a line status error occurs. Notice that when this interrupt is being serviced by the physical device driver in DMA mode, it is impossible to synchronize the line status event with the error character that caused the interrupt. The last character in the receive queue cannot be the error character, due to background DMA transfer activities occurring after the hardware interrupt. For precise synchronization of a line status error with a character, the Receive Data Status option is provided. With this option, for every byte of data that is received, a byte of status is received in the succeeding byte in the physical device driver receive queue.

Compared to conventional PIO-mode receive operations, DMA receive operations have different timing characteristics. As far as the timing of received data is concerned, the physical device driver is not notified through the receive interrupt mechanism, until one of the following events occurs:

- Up to 255 characters have been received.
- Receive buffer toggling has occurred.

Based on using the 80386 microprocessor REP MOVSD instruction:

- Receive FIFO timeout has occurred.
- Line status error occurred, when the physical device driver is not in Received Data Status mode.
- Character Compare Match interrupt occurred.

Notice that, due to these characteristics, DMA receive operation does not coexist with Multiple DOS Sessions-mode operations which, in many cases, are based on interrupt-driven and timing-sensitive operations.

DMA Receive Operation in OS/2 Protect Mode: The device driver uses two device driver receive queues, when operating in DMA mode, by toggling them during a receive operation. The size of each receive queue is 1KB, which makes the total size of the physical device driver receive queue 2KB. The physical device driver uses the pre-terminal count buffer transfer support provided by the ASYNC BIOS, which allows higher bit rate transfers without overruns. It provides a mechanism for receiving an interrupt prior to terminal count = 0, while still receiving data through DMA. The BIOS automatically toggles the queue by reloading the controller with the new address, when the end of the caller's queue is within reach of the number of paragraphs defined by the Receive Buffer Switch Threshold field of the BIOS interface. The BIOS then notifies the device driver, while DMA receive operation is performed in the other queue. The device driver can find out how many characters has been received at this moment, through the another BIOS function interface. The number for the Receive Buffer Switch Threshold is set by the device driver, and can vary, depending on the bit rate, to ensure successful buffer transition. With this option, the physical device driver is interrupted for a 255-character time frame, which is the maximum interval value supported by the hardware. This greatly reduces interrupt-driven system overhead, and frees the CPU, resulting in a significant performance increase throughout the system. This, in turn, allows the system to handle higher bit rate transfers.

Manipulating the two flat receive queues, demands new definitions for the buffer empty condition, buffer full condition, low water mark, and high water mark:

- The *buffer empty* condition occurs when both receive queues are empty.
- The *buffer full* condition occurs when one buffer reaches the end, and the other buffer is not fully empty.
- The *high water mark* condition occurs when one of the buffers reaches 60% of its capacity, while the other buffer is not fully empty.
- The *low water mark* condition occurs when one of the buffers is less than 70% full, while the other buffer is empty.

In non-Multiple DOS Sessions mode, the condition of the DMA receive operation is in either one of the following three states, which can be controlled by the user:

- DMA Receive Capability Enabled. (Default state.)
- DMA Receive Capability Disabled
- DMA Receive Capability Dedicated.

In *DMA Receive Capability Enabled* state, which is the initial system default, the physical device driver tries to allocate a DMA channel at Open request time for a receive operation. If the device driver fails to allocate a DMA channel, then it defaults to enhanced FIFO mode.

In *DMA Receive Capability Disabled* state, the device driver runs receive operations in enhanced FIFO mode.

In *DMA Receive Capability Dedicated* state, the device driver always runs receive operations in DMA mode.

If the user requests to switch the state of DMA Receive Capability, while a receive operation is in process, *there is a chance of loss of data*. To avoid this situation, the user is cautioned to check to see if the physical device driver receive queue is empty, through the ASYNC Generic IOCTL interfaces. In addition, communication protocols must be used to ensure the transmitting system on the other end of the line has stopped transmitting before requesting the DMA mode switch. When the user requests DMA Receive Capability Dedicated state, the device driver tries to allocate a DMA channel dedicated to receive operations for a COM port. If the physical device driver fails to allocate a dedicated channel, then it returns a *general failure* error. Once a dedicated channel is allocated, it is not deallocated until the user requests to switch to another DMA receive state. The state of a DMA receive operation set by the user does not change across Open requests.

In non-Multiple DOS Sessions mode, the physical device driver uses the Receive Line Status Interrupts option as a default that generates an interrupt on each line status error. This option performs a fast

operation of copying data from the physical device driver receive queue to the user buffer. With this mechanism, the precise association of a line status error, or a break event to a character cannot be achieved. Because of this characteristic, when the user requests the Error/Break Replacement Character option, the physical device driver switches to use the Receive Data Status option to ensure the correct Error/Break Replacement Character processing. If the user requests the Error/Break Replacement Character option, while the receive operation is in progress, there is a chance of data loss. It is recommended that the user have the transmitting system on the other end of the line stop transmitting data, as well as to check the physical device driver receive queue to ensure it is empty, before requesting the Error/Break Replacement Character option.

As seen from an application, the total size of the receive buffer in DMA mode is about 2KB, but because of the Receive Buffer Switch Threshold, the actual available size is less. This is different from conventional PIO mode, or enhanced FIFO mode, where it is about 1KB. The number of received characters is updated every time interval of 255-characters. This does not reflect the exact number of received characters at that time due to DMA characteristics.

Receive Operation in Multiple DOS Sessions Mode: Neither the DMA, nor the enhanced FIFO capabilities, are utilized by the physical device driver in Multiple DOS Sessions-mode receive operations. The device driver saves the current enhanced-mode state, and performs receive operations in conventional PIO mode. Upon termination of the Multiple DOS Sessions-mode operation, the previous state of enhanced receive mode is restored.

The DMA chip terminates the transmit operation, when the Terminal Count of the Enhanced UART reaches 0. This terminal count is set to the number of characters to be transmitted in the device driver transmit queue only at the start of the transmit interrupt, and reduced by 1 for each DMA character transfer. Because of these characteristics of the DMA operation, the physical device driver transmit queue is a flat queue.

The BIOS interface provides a way to stop and start the transmit operation, while it is running in DMA or enhanced FIFO mode. Using these features, the physical device driver can stop the transmit, send another character (such as an XON or XOFF character) immediately, and resume the original operation.

Compared to conventional PIO-mode transmit operations, DMA transmit operations have different timing characteristics. A DMA transmit request to an enhanced COM port returns immediately, as soon as the DMA chip moves the last character in the device driver's transmit buffer to the hardware FIFO buffer. At this moment, some of the characters in the FIFO buffer, including the last character, might not have been transmitted physically. In a conventional PIO-mode transmit operation, the physical device driver is informed by the UART, when the transmission is complete either on a per-character basis, or when the last character in the hardware FIFO buffer is transmitted.

Due to this timing characteristic, DMA transmit operation does not coexist with Multiple DOS Sessions-mode operations, which are written for PIO-mode operation, and are sensitive to the timing of individual character movement.

DMA Transmit Operation in OS/2 Protect Mode: The device driver uses a single flat transmit queue of 255 bytes in non-Multiple DOS Sessions mode. The length of the transmit queue, which is used as the Terminal Count, is set when the physical ASYNCR device driver starts a Write request by copying data from the user buffer to the physical device driver transmit queue. The Terminal Count is set again at the next transmit interrupt caused by the termination of the previous transmit operation, if the physical device driver needs to re-enable the transmit interrupt to handle the remainder of the requested data. This operation repeats until the requested number of characters to be transmitted becomes zero.

In non-Multiple DOS Sessions mode, the condition of the DMA transmit operation is in one of the following three states, which can be controlled by the user:

- DMA Transmit Capability Enabled. (Default state.)

- DMA Transmit Capability Disabled
- DMA Transmit Capability Dedicated.

In *DMA Transmit Capability Enabled* state, which is the initial system default state, the physical device driver tries to allocate a DMA channel at a Write request from the user. If the physical device driver fails to allocate a DMA channel, then it defaults to enhanced FIFO mode. Once the transmit operation starts to run in DMA mode, or enhanced FIFO mode, it stays in that mode until the write request from the user is satisfied.

In *DMA Transmit Capability Disabled* state, the device driver runs transmit operations in enhanced FIFO mode.

In *DMA Transmit Capability Dedicated* state, the device driver always runs transmit operations in DMA mode.

If the user requests to switch the state of DMA Transmit Capability, while a transmit operation is in process, *there is a chance of loss of data*. To avoid this situation, it is recommended that the user check to see if the transmit queue is empty, through the ASYNC Generic IOCTL interfaces, before requesting the DMA mode switch. When the user requests the DMA Receive Capability Dedicated state, the physical device driver tries to allocate a DMA channel dedicated to transmit operations for a COM port. If the physical device driver fails to allocate a dedicated channel, then it returns a general failure error. Once a dedicated channel is allocated, it is not deallocated until the user requests to switch to another DMA transmit state. The state of the DMA transmit operation set by the user does not change across the Open requests.

In non-Multiple DOS Sessions mode transmit/receive (on a COM port supported by the Enhanced UART or compatibles), when the physical device driver cannot allocate a DMA channel, it defaults to operate in enhanced FIFO mode unless the enhanced mode is turned off. In enhanced FIFO mode, the device driver not only exploits the efficiencies of the hardware FIFO buffer, but also takes advantage of the ABIOS function interfaces. Thus, compared to conventional PIO-mode operations, including the Extended Hardware Buffering capability, overall system performance is dramatically increased without sacrificing full compatibility. The Receive Trigger Level is set to 8, to avoid the possible hardware FIFO overrun, and the transmit operation uses a full 16-byte FIFO buffer. In this mode, the physical device driver ignores the Extended Hardware Buffering settings in Flags3 of the Device Control Block.

Except for the above characteristics, the operation is performed in the same way as in conventional PIO mode. There is a single circular transmit queue of 128 bytes, and a receive queue of 1KB.

Note: Compared to DMA mode, enhanced FIFO-mode operation has more interrupt-driven operation overhead. This limits the capacity to support high-bit-rate transmission. The physical device driver does not provide logic to limit the high bit rate in enhanced FIFO mode.

Transmit Operation in Multiple DOS Sessions Mode: Neither of the DMA, or the enhanced FIFO capabilities are utilized by the physical device driver in Multiple DOS Sessions-mode transmit operation. The device driver saves the current enhanced mode conditions, and performs a transmit operation in conventional PIO mode. Upon termination of the Multiple DOS Sessions-mode operation, the previous condition of enhanced transmit mode is restored.

RS232-C Enabling Characteristics in Enhanced Mode

An advanced set of functions, provided by the Enhanced UART or compatibles, facilitates the RS232-C communication protocols in DMA, or in Enhanced FIFO modes of operation. These functions allow the device driver to synchronize the communication protocol at the character level, without sacrificing the efficiency of DMA, or the full FIFO capacities of the hardware. Three types of the advanced features are utilized by the device driver:

Modem Pacing: The Modem Pacing capability allows the physical device driver to have the transmitter controlled through CTS, DSR, or DCD signals.

Character Compare Register Capability: Three character compare registers are provided for monitoring received characters. An optional action, on each character match of start transmitter, stop transmitter, delete character, or interrupt can be selected.

Start/Stop Transmission Functions: These commands enhance the data flow control in FIFO and DMA modes.

Input Modem Control Signals in Enhanced Mode

If the port is supported by the Enhanced UART or compatibles, the device driver automatically operates in enhanced mode, either in DMA mode, or in enhanced FIFO mode. In enhanced mode, the implementation of the following automatic communication protocol modes is performed through the advanced modem pacing features:

- Output Handshaking using CTS, DSR, DCD
- Input Sensitivity using DSR.

Logical Flow Control (XON/XOFF) in Enhanced Mode

Automatic Transmit Flow Control (XON/XOFF) operations in enhanced mode are performed through the use of the three *character compare* functions. With this option, when an XOFF character is received, the transmission operation stops. This XOFF character is automatically deleted from the hardware buffer, and an interrupt occurs to signal the physical device driver to record this event. When an XON character is received, the transmission is resumed. This XON character is automatically removed from the hardware buffer, and an interrupt occurs to signal the physical device driver to record this event.

Automatic Receive Flow Control (XON/XOFF) operations in enhanced mode are performed through the use of the Start/Stop Transmission BIOS functions. With this option, when the physical device driver receive queue reaches the high water mark, it issues the Stop Transmission and Send Data BIOS function. The transmission is stopped, and an XOFF character is sent. In the Normal mode of Automatic Receive Flow Control, the physical device driver does not send any more characters, and waits until the device driver receive queue reaches the low water mark. The physical device driver then sends an XON character by calling the Stop Transmission and Send Data BIOS function (at this moment, transmission has stopped; the physical device driver calls this function again to send an XON character), and calls the Start Transmission BIOS function to resume the previous transmit operation. In the full-duplex mode of Automatic Receive Flow Control, after sending an XOFF character, the physical device driver issues the Start Transmission BIOS function to resume the previous transmit operation. When the device driver receive queue reaches the low water mark, the physical device driver issues the Stop Transmission and Send Data BIOS function to send an XON character, and calls the Start Transmission BIOS function to resume the original transmit operation.

In Multiple DOS Sessions mode, the Stop Transmission and Send Data BIOS function is used in a similar way.

The "Function 44H – Transmit Byte Immediate" operation in enhanced mode is performed using the Start/Stop Transmission BIOS functions. The physical device driver issues the Stop Transmission and Send Data BIOS function to send the character, and issues the Start Transmission BIOS function to resume the original transmit operation. "Function 47H – Behave as if XOFF Received" and "Function 48H – Behave as if XON Received" operations in enhanced mode are performed by using the Start/Stop Transmission BIOS functions.

Bit Rate Considerations

The range of bit rates supported by the physical ASYNC device driver is from 2 bps (bits per second) to 19200 bps. Also, the clock frequency is 1.8432 MHz, and the error tolerance rate should be within $\pm .01\%$ (except 110 bps and 2000 bps, which are supported outside this error margin). To set or get bit rates, the ASYNC generic IOctls, "Function 41H — Set Bit Rate" on page 18-8, and "Function 61H — Query Bit Rate" on page 18-39 are provided.

The physical device driver has been directly programming the UART to set the bit rate, since the current ABIOS interface does not provide an interface to set a binary bit rate. The ASYNC physical device driver uses the new enhanced ABIOS function interfaces to set the bit rate. In this way, the clock frequency is transparent to the device driver. The supportability of the enhanced ABIOS function interfaces are determined at device driver initialization time. If the enhanced ABIOS interfaces are not present, then the advanced features of the Enhanced UART or compatibles, are not utilized by the physical device driver. If the enhanced ABIOS interfaces are available, then the physical device driver determines the highest, and lowest bit rates supportable for each COM port in the system at initialization time.

In enhanced mode, on a COM port with an Enhanced UART or compatible, and the enhanced ABIOS, the physical ASYNC device driver can support the range of bit rates from 10 bps to 345600 bps. The clock frequency used by the Enhanced UART is 22.1184 MHz. The user is advised not to depend on this clock frequency directly. The ABIOS provides a function interface to set a binary bit rate. When this function is called to set a bit rate, it returns with the actual physical value of the bit rate set. The ABIOS interface packet for the IOCtl Category 1 — Function 41H (Set Bit Rate) consists of a 24-bit field for the integer bit rate value, and an 8-bit field for the fraction of the bit rate value.

To support higher bit rates, the physical ASYNC device driver provides two new IOctls, "Function 43H — Extended Set Bit Rate" on page 18-11, and "Function 63H — Extended Query Bit Rate" on page 18-41. The user can use either Function 41H, or Function 43H, to set bit rates ranging from 10 bps up to 57600 bps for ports belonging to the Enhanced UART or compatibles. For bit rates higher than 57600 bps, the user has to use Function 43H. If Function 41H is used for bit rates higher than 57600 bps, the call fails with the *invalid parameter failure* return code.

For compatibility, the error tolerance rate of $\pm .01\%$ checking is also performed for any bit rates up to 19200 bps. To be compatible with 1.8432 MHz machine, the physical device driver uses the same logic as the one currently used to check the error tolerance rate for any bit rate up to 19200 bps. If the bit rate passes the error tolerance checking, then the user input bit rate value is checked for the valid range, and is passed to ABIOS, when the request was made through Function 43H. If the request is made through Function 41H, the physical ASYNC device driver manufactures the ABIOS interface value by putting the calculated resultant actual integer bit rate into the 24-bit integer field, zeroing out the fraction field, and passing it to ABIOS. If the bit rate checking fails, then the call fails with the *invalid parameter* return code.

For the bit rates higher than 19200 bps, the physical device driver does not check the error tolerance rate. At the request of "Function 43H — Extended Set Bit Rate," the physical device driver simply passes the binary user input to ABIOS, and saves the returned actual bit rate set. This saved actual bit rate is used as a returning value for Function 63H when the user queries the physical bit rate set. It is recommended that the user call Function 63H, after calling Function 43H, to examine the actual bit rate setting. It is the user's responsibility to handle the bit rate error tolerance rate for any bit rates higher than 19200 bps.

If a call is made through Function 61H for a COM port to get a bit rate higher than 0FFFFH bps, then the physical device driver sets the bit rate to 1200 bps (default), and returns 1200 to the user. If a call is made through Function 61H, and the bit rate is less than or equal to 0FFFFH bps, the physical ASYNC device driver returns the integer value of the current bit rate, and the fraction field is ignored, if it exists.

If the user calls Function 41H or Function 43H, for a COM port that is not on the Enhanced UART or compatible UART, the physical device driver supports only bit rates up to 19200 bps. The following table summarizes the supported bit rate ranges for Function 41H and Function 43H on COM ports, with conventional and enhanced serial communication devices. Notice that the bit rate achieved at the serial device level is not necessarily sustainable by the system.

Function	Hardware Port Type	Interface	Bit Range (BPS)
41H	Conventional	ABIOS	2 - 19200
41H	Enhanced (Enhanced UART)	ABIOS	10 - 57600
43H	Conventional	ABIOS	2 - 19200
43H	Enhanced (Enhanced UART)	ABIOS	10 - 345600

Supported Bit Rate Ranges

Control Panel Considerations

The number of COM ports supported by the Communications Port panel of the Control Panel has been increased to four ports.

Reserved Device Names (COM1-n)

The device name, AUX, does not appear in the device header of the ASYNCR device driver. The ASYNCR physical device driver does not support the reserved name, AUX, for either DOS applications, or OS/2 applications.

IBM PS/2 (with Micro Channel) Considerations for COM1-4

The IBM PS/2 ASYNCR device driver has device names, COM1, COM2, COM3, and COM4, in its device driver header. Device name, COM1, corresponds to the first LID in the ABIOS common data area with the ASYNCR Device ID. Device name, COM2, corresponds to the second LID. Device name, COM3, corresponds to the third LID. Device name, COM4, corresponds to the fourth LID in the ABIOS common data area with the ASYNCR Device ID.

The Advanced BIOS architecture ensures that the ordering in the ROM BIOS 40: data area matches the ordering of the LIDs in the common data area. Compatibility BIOS supports up to 4 ASYNCR devices on the IBM PS/2 system, and the physical device driver assumes that the order of the Logical IDs match the order of the addresses of these devices in the ROM BIOS 40: data area. Additional ASYNCR devices can be supported by additional device drivers.

The mapping of the ASYNCR Logical ID ordering to COMn, must be consistent across all device drivers that support the ASYNCR hardware in the IBM PS/2 hardware environment.

Initialization/Resource Management

The device driver is loaded and initialized with a DEVICE = statement in CONFIG.SYS. The physical device driver does not process any parameters on the DEVICE = statement. It is the responsibility of the installation process or the user to have the correct DEVICE = statement in the CONFIG.SYS file, depending on whether OS/2 2.0 is installed on:

- IBM Personal System/2* Models 57, 90, and 95, use COMDMA.SYS
- All others, use COM.SYS.

During initialization, the physical device driver attempts to free memory from its data segment for ports it does not need and that do not get initialized. The physical device driver does not remove device names from its device driver header for ports that do not get initialized.

The device driver does not deinstall a device, if the system requests it. If another device driver wishes to support a port already supported by this device driver, it needs to initialize *before* this ASYNC device driver.

Shared ownership of a given serial device between multiple device drivers in a single session of the OS/2 operating system is supported, subject to certain restrictions. Each driver that installs sharing a serial device obtains exclusive access to that device when it processes an Open request, rather than claiming the device during initialization. Each driver also fully relinquishes control of that device, when it processes a matching Close request.

When the physical device driver is initialized, it is enabled by default for Output Handshaking Using CTS and DSR. This is for compatibility with existing systems (IBM PC and PS/2 BIOS INT 14H) and the requirements of supporting an RS232 port. It is also enabled by default for Input Sensitivity to DSR. This allows a remote device to indicate whether data being received is valid, and is enabled to help ensure compatibility with existing systems. The initialization default for Extended Hardware Buffering on serial devices that fully support FIFO mode operations is Automatic Protocol Override.

Note: The ASYNC device driver default protocols provide an upwardly compatible RS232-C interface for communication with most devices. New communication applications and devices might not require the default protocols to be enabled. The user should evaluate these alternatives, and consider which protocols to enable or disable in order to provide the highest possible performance for support of a given serial device.

Initialization Considerations: The device driver does not attempt to initialize or support a port, if it does not get the INIT request packet for the port's corresponding device name. If the physical device driver gets the INIT request packet for a given device name, it checks to see if a valid I/O address is in the BIOS 40: data area that corresponds to that device name. COM1 is in 40:0 and COM2 is in 40:2. If the 40: area does not have a valid I/O address, the physical device driver fails the initialization of the port and will not support the port. Otherwise, the device driver attempts to get exclusively the interrupt level that corresponds to the I/O address for the port. If the interrupt level is not available, the physical device driver fails the initialization of the port and will not support the port.

If the interrupt level is available, the physical device driver relinquishes the interrupt level. The physical device driver also initializes the port, and sets up to support the port during this startup of the operating system.

In summary, in order for the physical device driver to support a port, the following must be TRUE:

- The device driver must get an INIT request packet for the device name.
- The 40: area that corresponds to the device name must have a valid I/O address.
- The appropriate interrupt level must be available for exclusive use, even though the physical device driver will not claim the interrupt level for exclusive use during initialization.

The physical device driver claims ownership of the port by not deinstalling the corresponding device name. Another device driver can cause this device driver not to claim a port by initializing *before* this device driver, and by doing at least one of the following:

* Trademark of the IBM Corporation

- Not allowing this physical device driver to receive an INIT request packet for a given device name
- Putting an invalid I/O address in the corresponding 40: area (for example, 0)
- Exclusively owning the appropriate interrupt level at initialization time.

The device driver does not attempt to initialize or support a port, if it does not get the INIT request packet for the port's corresponding device name. If the physical device driver gets the INIT request packet for a given device name, it attempts to claim ownership of the specific LID position for the ASYNC Device ID that corresponds to the device name being initialized.

For Micro Channel bus machines, if the LID is not available, the physical device driver fails the initialization of the port and does not support the port. If the LID is available, the physical device driver initializes the port and sets up support for the port during this startup of the operating system. The LID for the port is relinquished; it is reclaimed during Open processing.

For the AT bus machine, COM.SYS still install s even though the LID is not present.

In summary, for the physical device driver to support a port on IBM PS/2, the following must be TRUE:

- The physical device driver must get an INIT request packet for the device name.
- The ASYNC LID corresponding to the device name must be available.

The physical device driver claims ownership of the port by not deinstalling the corresponding device name. Another device driver can cause this device driver not to claim a port, by initializing *before* this device driver and doing at least one of the following:

- Not allowing this device driver to receive an INIT request packet for a given device name
- Claiming the appropriate ASYNC LID.

Data Translation/Monitor Support/Spooler Support

The physical device driver provides no data translation, code page, or monitor support. It is the responsibility of the application or subsystem to provide any function required in these areas.

Spooling from LPT to COM is supported by the spooler, but spooling from COM to LPT, or COM to COM, is not supported. When spooling from COM to LPT, code page support is provided by the spooler.

Any requests for registering or opening a monitor chain to COMn is rejected by the physical device driver. The physical device driver deals with binary data and provides no special processing of *binary* or *ASCII* data streams.

ASYNC Communication Device Driver Interfaces

The physical ASYNC device driver supports a large set of generic IOCTL functions (Category 1) that are organized into the following groups:

- Break Processing
- Device Control Block (DCB) Parameter Access
- Device Driver I/O Queue Management
- Line Characteristics
- Manual XON/XOFF Processing
- Polled Event Functions.

File System Requests

The physical ASYNC device driver supports the File System requests as shown in the following table:

Request	Description
INIT	Initialize the device
Open	Open the device
Close	Close the device
Read	Read from the device (Normal, Non-Destructive No-Wait)
Write	Write to the device
Status	Input or output status
Flush	Flush or terminate all pending requests.

Open Processing: The physical device driver does not claim the interrupt level the port is on until the port is open. If the interrupt level is not available, the OPEN request packet fails. The interrupt level is claimed exclusively on the ISA bus machines. The interrupt level is claimed shareable on the Micro Channel* architecture machines.

On PS/2 machines, a First Level Open causes the physical device driver to attempt to obtain a Logical ID. If this fails (which indicates another physical device driver might be using the device), a general failure is returned. If the Logical ID is obtained, but the open fails for some other reason, the Logical ID is freed. Other physical device drivers that co-exist with the OS/2 physical ASYNC device driver perform similar processing.

If a timer tick handler is not available during First Level Open processing, the Open request may fail. If the physical device driver receives an OPEN request packet and the COM device is not already open (from a previous open without a close), this is called a *First Level Open*, and the physical device driver does special processing. See "States of the ASYNC Device Driver" on page 8-8. If a subsequent Open request is issued before a previous First Level Open request has completed, the device driver might process the OPEN request packets in a different order than they were issued. This could cause the First Level Open to take effect at a time different from what the application expected.

An Open request should never be issued until a previous Last Level Close request has completed. Otherwise, the function performed by a Last Level Close and a First Level Open might not occur. If the port is not already open (First Level Open), the physical device driver attempts to clear out any data in the receive hardware. On the IBM PS/2 system, if the port is not already open (First Level Open), the physical device driver relies on the *Reset/Initialize Advanced BIOS* function to reset and clear the UART receive hardware.

Close Processing: An application should never close an open handle to a COM port while there are requests still pending for that handle. If a request has not completed, it might be waiting for timeout processing. IOCTLs are used to determine, and change, the current timeout processing.

A *Last Level Close* occurs when the port is no longer open (from another open without a close). When a Last Level Close occurs, the physical device driver does some special processing:

- Clears the receive and transmit queues.
- Turns break off, if currently transmitting a break.

* Trademark of the IBM Corporation

- Clears any character waiting to be transmitted immediately, if it cannot be transmitted. If it can be transmitted, the physical device driver makes sure that it is given to the transmit hardware.
- Attempts to automatically transmit an XON (if possible), if currently enabled for Automatic Receive Flow Control (XON/XOFF), and the last character that the physical device driver automatically transmitted was an XOFF.
- Waits until all the data in the transmit hardware has been physically transmitted.
- Unclaims the interrupt level.
- Turns DTR and RTS *off*, if they are not already *off*. The physical device driver first waits the specified number of character times.
- Relinquishes the Logical ID for the device (on PS/2 machines).

Read Processing: The physical device driver begins processing Read requests in the order it received them. Notice that this might not be the same order that the requests were issued by the application. If the physical device driver receives more than one Read request, the request is queued on the Read request queue for later processing. Applications might not see Read requests completed in the order that they were issued. The order of the data placed in the Read requests reflects the order the requests were received by the physical device driver.

The data for the Read requests comes from the physical device driver receive queue. Because of timeout processing, it is normal for the total number of Read characters requested not to be read. This is not considered an error. The request is completed when timeout is completed or when the amount of data requested is placed in the Read buffer. The various kinds of Read Timeout processing are discussed in the "States of the ASYNCPDD Device Driver" on page 8-8. To reduce the probability of a device driver *receive* queue buffer overrun, the communications protocol takes into account the size of the physical device driver receive queue.

Write Processing: The physical device driver begins processing Write requests in the order it received them. Notice that this might not be the same order that the requests were issued by the application. If the physical device driver receives more than one Write request, the request is queued on the Write request queue for later processing. Applications might not see Write requests complete in the order they were issued. The order of the data transmitted due to the Write requests reflects the order that the requests were received by the physical device driver.

The data from the Write requests are placed in the physical device driver transmit queue. The number of characters written is considered to be the number of characters given to the transmit hardware, and not the number of characters placed in the physical device driver transmit queue. Because of timeout processing, it is possible that the total number of Write characters requested will not be transmitted. This is not considered an error. The request is completed when it times out or when the amount of data requested is given to the transmit hardware (but not actually transmitted at the physical RS232-C interface). Write Timeout processing is discussed in "States of the ASYNCPDD Device Driver" on page 8-8.

If infinite Write Timeout processing is enabled, it is the responsibility of the application to monitor the status of the Write requests. The application might have to issue an `IOCTL` to disable infinite Write Timeout processing to cause the Write request to complete (without all the data being transmitted). If an application does not check that all the data is given to the transmit hardware on each Write request, use the infinite timeout processing mode of the physical device driver to ensure that all the data has been given to the *transmit* hardware before the request completes. To increase the throughput (ratio of number of characters transmitted per second to the bit rate), the application should keep the Write requests as large as possible.

Access Authorization

The physical ASYNC device driver does not prevent multiple processes from concurrently opening the same device name. The physical device driver uses standard file system contention control. An application or subsystem, which needs exclusive access to the COM device, will open it with access rights set to DENY_ANY. Inheritance of open handles, and sharing of the device among multiple opens, work in a manner similar to regular files.

ASYNC (RS232-C) Generic IOCTL Command Summary

The following table lists each function and its description:

Function	Category 1 IOCTLs
	Line Characteristics
41H	Set Bit Rate
42H	Set Line Characteristics (stop, parity, data bits)
43H	Extended Set Bit Rate
46H	Set Modem Control Signals
61H	Query Current Bit Rate
62H	Query Line Characteristics
63H	Extended Query Bit Rate
66H	Query Modem Control Output Signals
67H	Query Current Modem Input Signals
	Manual XON/XOFF (Flow Control) Processing
44H	Transmit Byte Immediate
47H	Behave as if XOFF Received (stop transmit)
48H	Behave as if XON Received (start transmit)
	Break Processing
45H	Set Break OFF
4BH	Set Break ON
	Device Driver I/O Queue Management
68H	Query Number of Characters in Receive Queue
69H	Query Number of Characters in Transmit Queue
	Polled Events
64H	Query COM Status
65H	Query Transmit Data Status
6DH	Query COM Error
72H	Query COM Event Information
	Enhanced Parameters
54H	Set Enhanced Mode Parameters
74H	Query Enhanced Mode Parameters
	Device Control Block (DCB) Parameter Access
53H	Set Device Control Block Parameters
73H	Query Device Control Block Parameters
	Device Control Block parameters determine:
	<ul style="list-style-type: none"> • Automatic Transmit Flow Control (start/stop transmit, when XON/XOFF received) • Automatic Receive Flow Control (transmit XON/XOFF, when receive buffer fills/empties) • XON/XOFF Characters • DTR Control Mode (enable/disable/input handshaking) • RTS Control Mode (enable/disable/input handshaking/toggling on transmit) • Output Handshaking Using CTS/DSR/DCD (control signal determines when to transmit) • Input Sensitivity Using DSR (reception of data controlled by DSR) • Error Replacement Character and Processing • Break Replacement Character and Processing • Null Stripping • Timeout Processing • Extended Hardware Buffering (DISABLED/ENABLED/Automatic Protocol Override).

See Chapter 18, "Generic IOCTL Commands" on page 18-1 for detailed descriptions of the ASYNC Category 1 IOCTL functions.

DOS Session Considerations/Restrictions

It is strongly recommended that applications using a serial printer from a DOS Session, spool print data to the spooler through the appropriate LPT handles.

The physical device driver makes no attempt to restrict or mold the function of file system requests, because they might have come from a DOS Session. To achieve the full capabilities of the file system access to COM, the application needs access to the full range of Category 1 IOCTLs. The design and externals of the physical ASYNC device driver are based on the requirements of OS/2-mode applications that use the RS232-C port of the system.

Control Panel Considerations

When setting up a serial printer in the Control Panel, the user can choose between two handshaking protocols. If the user selects **None**, the serial printer card switches must be set for XON/XOFF handshaking. If the user chooses **Hardware**, the serial printer card switches must be set for DTR Pacing.

Performance

The achievable performance is very sensitive to the environment. The type and amount of other system activity determines the achievable performance. On the IBM PS/2 system, the number of COM ports or other devices on the same interrupt level, significantly affects the achievable performance level.

Trying to receive data at too high a bit rate could cause *hardware overrun errors*, or *receive queue overrun errors*. Receive queue overrun errors are easily solved by adjusting the communications protocol to the size of the physical device driver receive queue. Trying to transmit data at too high a bit rate could also cause the performance of the operating system to be reduced severely.

The bit rate can be set with the MODE command, or with an IOCTL. The bit rate should not be set to values which might cause receive overruns or adverse OS/2 system performance effects.

Enabling Extended Hardware Buffering: In most transmit-only situations (for example, serial printers), there is always a requirement for flow control (using Output Handshaking or Automatic Transmit Flow Control). If the attached hardware can receive a significant number of characters after the modem control (pacing) signal is changed, then setting Extended Hardware Buffering to enabled (Category 1 – Function 53H) can be an acceptable way to significantly improve the transmit throughput, and improve the operating system performance. This allows the Extended Hardware Buffering to yield maximum serial I/O performance while still providing the required Output Handshaking, or Automatic Transmit Flow Control protocols required by the attached serial device. Testing with Extended Hardware Buffering enabled, must be performed at the attached device when the physical ASYNC device driver is placed in this mode.

In many Receive and Transmit (half or full duplex) scenarios, disabling Input Sensitivity Using DSR has no negative effects. Many communications devices have switches, which cause DSR to be constantly *on*. Disabling Input Sensitivity Using DSR significantly improves the ability of the serial port hardware to handle receive data without receive hardware overrun errors. Another possible approach is to set Extended Hardware Buffering enabled by using the Set Device Control Block Information IOCTL function, or the OS/2 MODE command.

In some other Transmit and Receive scenarios, disabling Output Handshaking Using CTS and DSR after a communications link has been established, has no adverse effects under normal operating conditions.

Again, the achievable performance benefits can be significant. Another possible approach is to set Extended Hardware Buffering enabled by using the Set Device Device Control Block Information IOCTL function.

The potential negative effects of disabling a default control mode of the physical device driver should be thoroughly understood. The potential performance benefits, however, can far outweigh the added complexity of usage, and any other potential problems. Such problems can usually be resolved either by reverting to the Automatic Protocol Override mode, or by setting Extended Hardware Buffering to disabled by using the Set Device Control Block Information IOCTL function.

In weighing the alternatives, the per-character processing requirements must be addressed when deciding whether to override one of the default protocols of the physical device driver. Possible data integrity problems, such as receive overrun errors, loss of data at the beginning or end of a communications session, or receiving invalid data on a noisy communications connection, can occur. Such problems must be considered before using the potential performance benefits associated with Extended Hardware Buffering.

For ports operating at a given data transfer rate, the system has less than 20% interrupt-driven device driver overhead when running with Extended Hardware Buffering enabled (both transmit and receive FIFO buffering active), as compared with running those ports on devices where Extended Hardware Buffering is disabled.

Also of equal importance are the operational characteristics of the device driver. By setting Extended Hardware Buffering enabled, the physical device driver can transmit with the full 16-character FIFO buffer active (essentially transmitting 16 characters at a time), and the Receive Data Available interrupts can provide 4, 8, or 14 characters each to the physical device driver receive queue. This is true no matter what device driver protocols are enabled. Examples of scenarios where setting the FIFO Enabled mode of the physical device driver might be acceptable are:

- If the user does not care if too many characters are transmitted after a modem connection is broken
- If the printer or plotter connected to the system does not lose data when it tells the system to stop transmitting with a modem control signal, and the system continues to transmit a significant number (up to 16) of characters.
- If the system is connected to a modem or another system that is normally set up to always keep DSR on.
- If the communications protocol with the remote device does not require the system to respond on a character-by-character basis to input data. For example, when the remote device sends data in blocks and provides *error retry* capability on a block basis rather than a per-character basis.

Chapter 9. Physical CLOCK\$ Device Driver

The OS/2 operating system assumes that a CMOS real-time clock is available in the system. The CLOCK\$ device defines and performs functions like any other character device, except that it is identified by a bit in the attribute WORD. The operating system uses this bit to identify the device driver; therefore, this device can take any name. The device has been named *CLOCK\$* to avoid possible conflicts with any files named *CLOCK..*

The CLOCK\$ device is unique, because the OS/2 operating system reads or writes a 6-byte sequence, which encodes the date and time. Writing to this device sets the date and time; reading from it gets the date and time. The following figure illustrates the binary time format used by the CLOCK\$ device:

Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
Days since 1-1-80 Low-byte Hi-byte	Minutes	Hours	Sec/100	Seconds	

Figure 9-1. CLOCK\$ Device Time Format

The physical CLOCK\$ device driver sets and maintains the following fields in the global InfoSeg:

TIME

- Time, from 1-1-1970, in seconds
- Milliseconds
- Hours
- Minutes
- Seconds
- Hundredths
- Timer interval.

DATE

- Day
- Month
- Year
- Day of week.

The physical CLOCK\$ device driver ensures that the date, time in seconds (from 1-1-1970), and time of day (hours, minutes, seconds) fields remain synchronized with the CMOS clock and that the hundredths of seconds field is correctly synchronized with the seconds field.

Chapter 10. Physical Fixed Disk and Diskette Device Driver

The physical Fixed Disk and Diskette device driver provides access to disk and diskette data and controls removable diskette media. The physical device driver runs in a multi-tasking, protect-mode environment. The physical device driver receives the generic Category 8 and 9 IOCTL commands from the kernel, processes them to completion, and then returns status:

The device driver's physical devices are represented to the kernel as a set of logical units following the Extended DOS Partition Architecture. The physical device driver presents a logical unit as a sequence of 512 byte sectors.

The kernel and the physical device driver maintain removable volume integrity by making sure that the correct diskette is mounted. The physical device driver informs the kernel of media changes through the return code. The kernel can then issue interactive prompts to request the appropriate media to be mounted.

The physical characteristics of logical units are described in BIOS Parameter Block (BPB) control blocks, maintained jointly by the device driver and the kernel. The BPB contains the size and physical geometry of the logical unit, and file system fields that are not used by the device driver.

Generic Category 8 IOCTL functions access the logical unit using its cylinder, head, and sector geometry. Generic IOCTLs bypass the file system control of the logical unit, and are intended for system applications that set up file system environments. Generic Category 9 IOCTL functions access physical units.

Configuration

The IBM PS/2 ABIOS physical Disk device driver supports up to three internal diskette drives (1.44MB and 2.88MB), and one external 5.25 inch 360KB diskette drive. Fixed disk support is provided for up to 24 physical disk drives configured as ABIOS disk Logical IDs (LIDs). Up to 24 units are supported in any unit-LID combination. Three shared hardware interrupt levels are supported; one for diskette and two for fixed disk LIDs. Co-residency of multiple disk adapters is also supported.

Performance Considerations

All requests are queued by the device device internal queue management routines. Requests are sorted based on an algorithm that minimizes seek distance, within a given request priority.

For the IBM PS/2, independent disk LIDs and units on concurrent disk LIDs are programmed for overlapped I/O operations by the device driver.

EXTDSKDD.SYS Support

This device driver defines additional drive letters for currently installed diskette drives on a computer. The drive parameters can also be set to default to a lower capacity drive. For example:

```
DEVICE=C:\OS2\EXTDSKDD.SYS /D:0 /F:720KB
```

This defines a new drive letter, **D:**, for the (1.44MB/2.88MB) diskette drive 0 (**A:**), which acts like a 720KB diskette drive.

DOS Extended Volume Architecture

Fixed disks can be divided into primary partitions, and an extended partition that contains multiple logical block devices. The *extended partition* is indicated by a System ID byte of 05H in the partition table of the Master Startup (Boot) Record. This partition cannot be started, and programs that can set startable partitions (such as OS/2 FDISK) do not allow the partition to be marked as able to start.

Note: This extended partition support can be used on any fixed disk supported by the OS/2 operating system.

The extended DOS partition can be created only if a primary DOS partition already exists on a startable drive. A *primary partition* is a partition with a System ID byte of 01H, 04H, 06H, or 07H. If the drive cannot be started, then an extended DOS partition can be created without having a primary DOS partition.

Note: FDISK refers to extended volumes as *logical drives*.

The extended DOS partition starts and ends on a cylinder boundary, and contains a collection of extended volumes that are linked together by a pointer in the extended volumes' extended startup record. An *extended volume* consists of an extended startup record and one logical block device. In OS/2 Version 1.0, an extended volume cannot be larger than 32MB, due to the limitations of the FAT file system. However, in OS/2 2.0, this restriction has been removed. An extended volume created within the extended DOS partition can be any size, from one cylinder long through the maximum available contiguous space in the extended DOS partition. An extended volume can be larger than 32MB. All extended volumes must start and end on a cylinder boundary. An extended volume corresponds to an image of a physical disk. The extended startup record corresponds to the Master Startup Record at the beginning of an actual physical disk. The logical block device corresponds to the DOS partition that is pointed to by the Master Startup Record.

The logical block device begins with a normal DOS startup sector if it is a DOS logical block device (System ID=1, 4, or 6). Installable File System (IFS) logical block devices (System ID=7) need not start with a normal DOS boot sector. This logical block device must start on a cylinder and head boundary, and follow the extended startup record on the physical disk. The logical block device and the extended volume both end on the same cylinder boundary.

Each extended volume contains an extended startup record located in the first sector of the disk location assigned to it. This extended startup record contains the 55AAH signature ID byte. This allows programs that look at the Extended (Master) Startup Record to be compatible. This extended startup record also contains a partition table, which can contain only two types of entries. The startup code is not critical, as the devices are not considered startable. The startup code can simply report a message indicating an unstartable partition if it is executed.

The partition table portion of the extended startup record is the same as the partition table structure in the Master Startup Record. This structure has four partition entries of 16 bytes each. The System ID byte must be filled in for all four entries with one of the following values:

- 00H** No space allocated in this entry.
- 01H** DOS partition up to 16MB.
- 04H** DOS partition with 32MB > SIZE > 16MB.
- 05H** Maps out area assigned to the next extended volume. Serves as a pointer to the next extended startup record.
- 06H** DOS partition size > 32MB.
- 07H** Installable file system.

If the System ID byte is 0, then the values in that partition table entry are set to 0.

If the operating system detects any values other than 01H, 04H, 06H, or 07H, it ignores that entry, and does not attempt to install the logical block device. This allows future expansion of devices in this area without problems of compatibility with earlier systems.

The partition start and end fields, Cylinder, Head, and Sector (C,H,S) are filled in for any of the four partition entries in an extended startup record that have one of the above System ID bytes. This allows a program such as FDISK to determine the allocated space in the extended DOS partition, and allows the physical device drivers to determine the physical DASD area that belongs to it. The partition start and end fields (C,H,S) for the partition entry that points to the logical block device (System ID 01H, 04H, 06H, or 07H) map out the physical boundaries of the logical block device. They are offset relative to the beginning of the extended startup record that the entry resides in. The partition start and end fields for the partition entry that points to the next extended volume (System ID 05H), map out the physical boundaries of the next extended volume. They are relative to the beginning of the entire physical disk.

The relative sector and number of sector fields are set up differently depending on what System ID byte is used. If 01H, 04H, 06H, or 07H is in the System ID field for that extended partition entry (pointer to the logical block device), the relative sector field is set up as an offset from (and including) the start of the extended startup record for the associated extended volume. The *number of sectors* field is filled in with the size of the created logical block device area, that is, the number of sectors mapped out by the start and stop cylinder/track/sector fields. The size of the extended volume can be calculated by adding the relative sector field and the sector size field of the associated extended startup record.

If the System ID byte is 05H, then the relative sector field is the offset (of the next extended volume) in sectors from the start of the entire extended DOS partition. The number of sectors field is not used in this field, and is filled with 00Hs.

This architecture allows only one logical block device to be defined for each extended startup record. Therefore, only a maximum of two partition entries at a time is used in each extended startup record; an entry with System ID byte of 01H, 04H, 06H, or 07H, and an entry with ID of 05H (which is the pointer to the next extended volume). Although only two entries can be used, a program installing these devices does not assume that the first two entries will be the non-zero entries.

Installing Block Devices in the DOS Extended Partition

To install block devices, physical device drivers first install the primary DOS partitions on all physical drives, if any exist. This insures that an existing drive letter, **D:**, on the 81H drive remains the same. After these devices are installed on the 80H drive, the drivers look for the existence of the extended DOS partition. If one exists, then the physical device drivers look at the first sector of the extended DOS partition for the first extended startup record. If there is a valid System ID (01H, 04H, 06H, or 07H) in any of the four partition entries, the device is installed and assigned the next available drive letter. This occurs before any CONFIG.SYS device drivers are loaded, so the FDISK will correctly display the drive letter when space is allocated for the drive.

The first extended startup record (in the extended DOS partition) is a special case, because it is possible there will not be a device to be installed defined in the partition table. The first device might have been created and then deleted at some time. However, the first extended startup record is needed to point to the next one, if one exists. Any other extended startup record will always have a device to be installed.

Once a device has been installed (or the special cases above occur), the physical device driver searches the other partition entries for a System ID byte of 05H, indicating that another device (extended volume) exists. If a 05H is not found, there are no more logical block devices (extended volumes) in the extended DOS partition.

If a 05H System ID is found, the start location in that partition entry is read in order to find the location of the next extended startup record. When located, it is read in, and then the process is repeated in order to install additional devices.

Once all the valid devices for a physical drive have been installed, the next physical drive is examined and the entire process is repeated.

A device driver does not assume any order dependency when searching for a particular System ID byte in an extended startup record. All four possible entries in a extended startup record partition table are searched, before a driver decides that a particular System ID byte does not exist.

The extended DOS partition can only be created if a primary DOS or IFS partition already exists on a bootable drive. A *primary DOS partition* has a System ID of 01H, 04H, or 06H. A *primary IFS partition* has a System ID of 07H. If the driver is not bootable, an extended DOS partition can be created without having a primary DOS partition. The extended DOS partition starts and ends on a cylinder boundary.

Creating Block Devices in the Extended DOS Partition

To create the structure for an extended volume in the extended DOS partition, FDISK determines if there is available space in the extended DOS partition, and if less than 24 total devices are allocated in the system. The maximum number of block devices allowed is 26, and two are used by diskettes, **A:** and **B:**. The program then creates an extended startup record at the space located, with a partition entry filled in (with the size and location information) for that logical block device. If this is not the first extended startup record, the program backs up to the last extended startup record in the chain (as linked by the 05H entries), and creates a partition entry in that extended startup record that has the size and location data for the newly created record. This action creates the pointer required to locate the newly created startup record.

If this is the first extended startup record in the extended DOS partition, only the size, type and location of the logical block device needs to be put into a partition entry. The start of the extended DOS partition in the Master Startup Record serves as a pointer to this extended volume.

Deleting Block Devices in the Extended DOS Partition

To delete a block device, the program sets the 16-byte partition entry that contained the System ID byte, to 0. If in the same extended startup record, there exists a partition entry with System ID of 05H, indicating that another extended volume exists, this information is copied to the 05H partition entry of the previous extended startup record.

Note: There is one exception to this rule. If the deleted logical block device is at the beginning of the extended DOS partition, only the partition entry indicating the device type is set to 0. The 05H pointer information is be left in place.

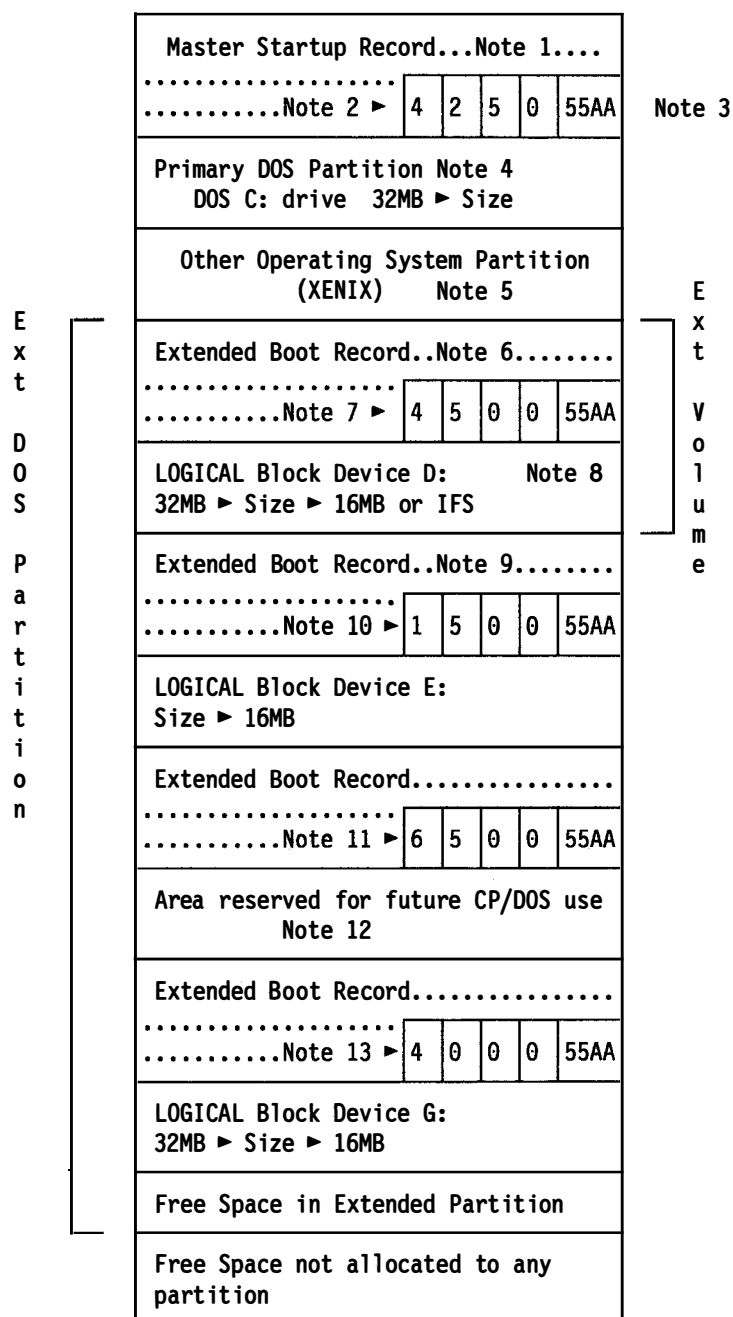


Figure 10-1. Layout Example of Large DASD Device with Extended DOS Partition.

Note 1 Master Startup (Boot) Record code, starting at Track 000, Head 00, Sector 01 of disk 80H or 81H.

Note 2 Partition table for Master Startup Record. See "BPB and Get Device Parameters for Extended Volumes" on page 10-7 for the layout. The 4 is the System ID byte in the partition table that indicates a DOS partition greater than 16MB, and less than or equal to 32MB. The 2 is a XENIX¹ partition, and the 05H maps the extended DOS partition.

Note 3 55AAH is the signature to validate the Master Startup Record.

¹ XENIX is a trademark of the Microsoft Corporation

- Note 4** Primary DOS area, must reside entirely in first 32MB of disk. **C:** is block device 80H. **D:** is block device 81H, if it exists. This partition has a maximum size of 32MB.
- Note 5** Other operating system on disk.
- Note 6** Extended startup record for extended volume that corresponds to logical block device **D:**. (This assumes only the 80H block device exists.) If 81H block device exists, this would be block device **E:**.
- Note 7** Logical block device **D:** partition table entry. This has a maximum size of 32MB, which is indicated by the System ID of 4. This must set the logical DOS block device as starting at the next track boundary. The 05H System ID byte in the second partition entry maps out the space allocated to the next extended volume. The starting cylinder/sector/head in the partition entry with ID of 05H is the location of the next extended startup record of the next extended volume.
- Note 8** Logical block device **D:**. Logical DOS devices and the primary DOS partition always begin with a DOS startup record.
- Note 9** Extended startup record for logical block device **E:**.
- Note 10** Partition table entry for logical block device **E:**. This logical DOS block device is less than or equal to 16MB, as indicated by the System ID of 01H. The entry with System ID of 05H maps out the space allocated to the next extended volume.
- Note 11** The System ID byte of 06H indicates a logical block device greater than 32MB. This block device is indicated by a block device letter of **F:**. Note also that a pointer exists to the next extended volume.
- Note 12** The greater than 32MB FAT partition.
- Note 13** Partition table entry for final DOS logical block device. Note the absence of the 05H ID byte means that there are no other extended volumes allocated in the extended DOS partition. This would have a block device letter of **G:**, if the previous logical block device was recognized. Otherwise it would be **F:**.

Offs Purpose	Head	Sector	Cylinder	
1BE Partition 1 begin	boot ind	H	S	CYL
1C2 Partition 1 end	syst ind	H	S	CYL
1C6 Partition 1 rel sect	Low word		High word	
1CA Partition 1 # sects	Low word		High word	
1CE Partition 2 begin	boot ind	H	S	CYL
1D2 Partition 2 end	syst ind	H	S	CYL
1D6 Partition 2 rel sect	Low word		High word	
1DA Partition 2 # sects	Low word		High word	
1DE Partition 3 begin	boot ind	H	S	CYL
1E2 Partition 3 end	syst ind	H	S	CYL
1E6 Partition 3 rel sect	Low word		High word	
1EA Partition 3 # sects	Low word		High word	
1EE Partition 4 begin	boot ind	H	S	CYL
1F2 Partition 4 end	syst ind	H	S	CYL
1F6 Partition 4 rel sect	Low word		High word	
1FA Partition 4 # sects	Low word		High word	
1FE Signature				

Figure 10-2. Partition Table for Master Startup Record

BPB and Get Device Parameters for Extended Volumes

For purposes of the BIOS Parameter Block (BPB) and Get Device Parameters (generic IOCTL), an extended volume appears to the system as a fixed disk. The extended startup record corresponds to the Master Startup Record of a real fixed disk and the logical block device corresponds to the primary DOS partition.

This means that the BPB of the logical DOS block device of the extended volume describes the environment in the extended volume. This consists of the extended startup record and the logical block device. The meaning of the fields is consistent with the meaning of the fields for the primary DOS partition; they relate to the entire physical disk, the primary DOS partition, and the Master Startup Record. For example, the number of hidden sectors is the distance from the beginning of the extended startup record (of the extended volume in question) to the start of the logical DOS block device (the DOS Startup Record). The number of sectors field describes only the logical block device, just as it normally only describes the primary DOS partition.

Category 8 Generic IOCTL Commands

The philosophy described above also applies to the disk generic IOCTL commands. For any logical block device of an associated extended volume; physical cylinder, head, and sector I/O is mapped to within the extended volume. Cylinder 0, Head 0, Sector 1 is mapped to the extended startup record. An error condition is generated for any attempt to do C,H,S I/O beyond the size of the extended volume in question.

Category 9 Generic IOCTL Commands

Category 9 generic IOCTL commands are used to access the entire physical fixed disk without consideration of logical volumes. Physical cylinder, head, and sector begin at the start of the physical drive, instead of at the beginning of an extended volume.

Type 6 Partition

A 12-bit or 16-bit type FAT can be used to map a Type 6 partition since the type of FAT is strictly based on the number of allocation units (clusters), and is the same algorithm used to define the type of FAT in the OS/2 Version 1.0 operating system. FAT cluster sizes are based on powers of 2. Assuming usage of the OS/2 FORMAT utility, the minimum cluster size for a hard file is 2KB. Cluster size and the type of FAT (12-bit versus 16-bit) is determined by the media partition size. The OS/2 FORMAT algorithm is:

```
If partition size <= 16MB
then;
    use 12-bit FAT;           /* max 4084 entries */
    max cluster size = 4KB;
end;
else;
    use 16-bit FAT;           /* partition size > 16MB */
    /* max 64KB entries */
    min cluster size = 2KB;
end;
```

The actual determination of the partition type is made based on the number of clusters on that partition. OS/2 FORMAT makes sure that this is true for the < 16MB and > 16MB partitions.

```
If number of clusters <= 4084
    use 12-bit FAT;           /* max 4084 entries */
else
    use 16-bit FAT;           /* max 64KB entries */
```

A partition size of 128MB requires a 2KB cluster size, based on a maximum of 64KB allocation units (clusters). A partition size of 129MB – 256MB requires a 4KB cluster size, based on 64KB allocation units. A partition size of 257MB – 512MB requires an 8KB cluster size, based on 64KB allocation units.

The configuration table used by OS/2 FORMAT is as follows:

Total # of Sectors	Size of Partition	Sector Cluster	3 of Root DIR Entries
32K	16M	8	512
64K	32M	4	512
256K	128M	4	512
512K	256M	8	512
1M	512M	16	512
2M	1G	32	512
4M	2G	64	512
8M	4G	128	512

OS/2 FORMAT Configuration

Note: For Type 6 partitions, it is safe to use a non-default configuration, but this may be unsafe for other partition types.

The partition can reside anywhere on the media, as the primary DOS partition, or as an extended volume within the extended DOS partition. The BPB parameter *number of sectors per FAT* field width has been extended from a byte to a WORD, in order to define a full 128KB FAT structure. This change affects all DOS partition types.

In the extended volume context, Type 6 > 0MB. In the primary DOS partition context, Type 6 > 0MB. If the partition spans the 32MB boundary, or starts at or beyond the 32MB boundary, it can be any size > 0, and still be a Type 6 partition. If the media is > 32MB, or is defined by a Type 6 partition, the recommended BPB must show a 0 in TotalSectors. Ext_TotalSectors must show the number of sectors on the media, and the doubleword form of *HiddenSectors* must show the number of sectors after the Master Startup (Boot) Record to the start of the partition's OS/2 boot record. See "Function 63H – Query Device Parameters" on page 18-159 for the parameter information.

Layout of Block Devices with a Type 6 Partition

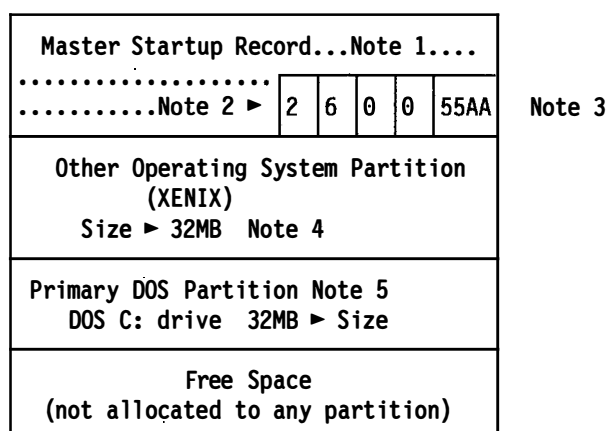


Figure 10-3. Layout Example of Large DASD Device with a Type 6 Partition.

- Note 1** Master Startup (Boot) Record code, starting at Track 000, Head 00, Sector 01 of disk 80H or 81H.
- Note 2** Partition table for Master Startup (Boot) Record. The 2 is the System ID byte in the partition table that indicates a XENIX partition, and the 06H maps indicate a primary DOS Type 6 partition.
- Note 3** 55AAH is the signature to validate the Master Startup (Boot) Record.
- Note 4** Other operating system (XENIX) on disk.
- Note 5** Primary DOS partition. C: is block device 80H. The partition type in this example is a 6, because it ends beyond the first 32MB of the disk. Within the scope of this definition, though the size of a primary DOS partition can be less than 32MB (because it ends beyond the first 32MB of the disk), it is defined as a Type 6.

Layout of Block Devices with a Type 6 Partition

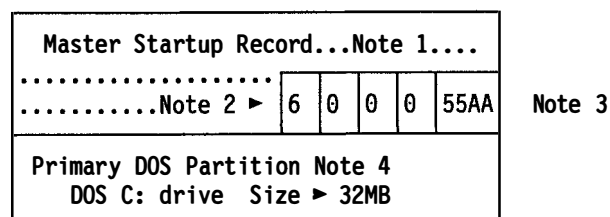


Figure 10-4. Example of Layout of Large DASD device with a Type 6 Partition.

fixed disk/diskette PDD

- Note 1** Master Startup (Boot) Record code, starting at Track 000, Head 00, Sector 01 of disk 80H or 81H.
- Note 2** Partition table for Master Boot Record. The 6 is the System ID byte in the partition table that indicates a DOS partition where 32MB < SIZE.
- Note 3** 55AAH is the signature to validate the Master Startup (Boot) Record.
- Note 4** Primary DOS area. Owns the entire media and exceeds 32MB in size. **C:** is block device 80H.

Type 7 Partition

Partition Type 7 is used for Installable File Systems only. The internal FAT file system should not use this partition type since it means that older versions of PC-DOS and OS/2 operating systems will not be able to access the partition.

Chapter 11. Physical Keyboard Device Driver

In OS/2, the generic console device driver has been replaced by two independent physical device drivers:

- Keyboard input (KBD\$)
- Screen or console output (SCR\$).

The physical KBD\$ device driver supports the OS/2 interrupt-driven architecture. The physical SCR\$ and KBD\$ device drivers are part of the resident physical device driver set.

Keyboard Input (KBD\$)

The physical Keyboard device driver receives the make-and-break keystroke scan codes along with special keyboard hardware codes, and performs one or more of the following operations:

- Translates the scan code to an ASCII character
- Recognizes the key as a special key and signals the appropriate routine for processing
- Passes the translated scan code's character data record to the monitor dispatcher for further custom processing by monitors
- Places the translated scan code's character record into the appropriate session's keyboard input buffer.

In addition, the physical Keyboard device driver supports code page switching. The following is a result of this support:

- Each handle can use one of two system-wide code pages. One code page is current; the other can be switched to. Each handle can also use the PC US 437 code page.
- The code pages used are defined in CONFIG.SYS with the CODEPAGE and DEVINFO commands.
- Code pages specified with the CODEPAGE command in CONFIG.SYS are for one language only. However, it is possible that code page support for different languages can result, if default code pages are not accepted during execution of the KEYB command. This can occur when attempting to load translate tables in a non-supported code page for that language.
- The user is allowed, through the KEYB command, to change the language layout of the keyboard.
- The user can control, by the KbdSetCp subsystem function or the CHCP command, which of the two code pages is used in the translation of scan codes.
- Two subsystem functions support code page switching:
 - KbdSetCp - Set to an installed code page, load if necessary
 - KbdGetCp - Get the current in-use code page.
- KbdSetCustXt adds a custom code page option.
- KbdXlate translates a scan code.
- KbdSetCp, KbdGetCp, and KbdXlate can be replaced in the subsystem by the use of the KbdRegister function.
- Code page number, Language ID of the translation table, Subcountry ID, language default table indicator, and keyboard type indicator can all be found in the Translation table header of each country's keyboard layout. See IOCTL Category 4 "Function 50H — Set Code Page" on page 18-66 for further keyboard translation table details.

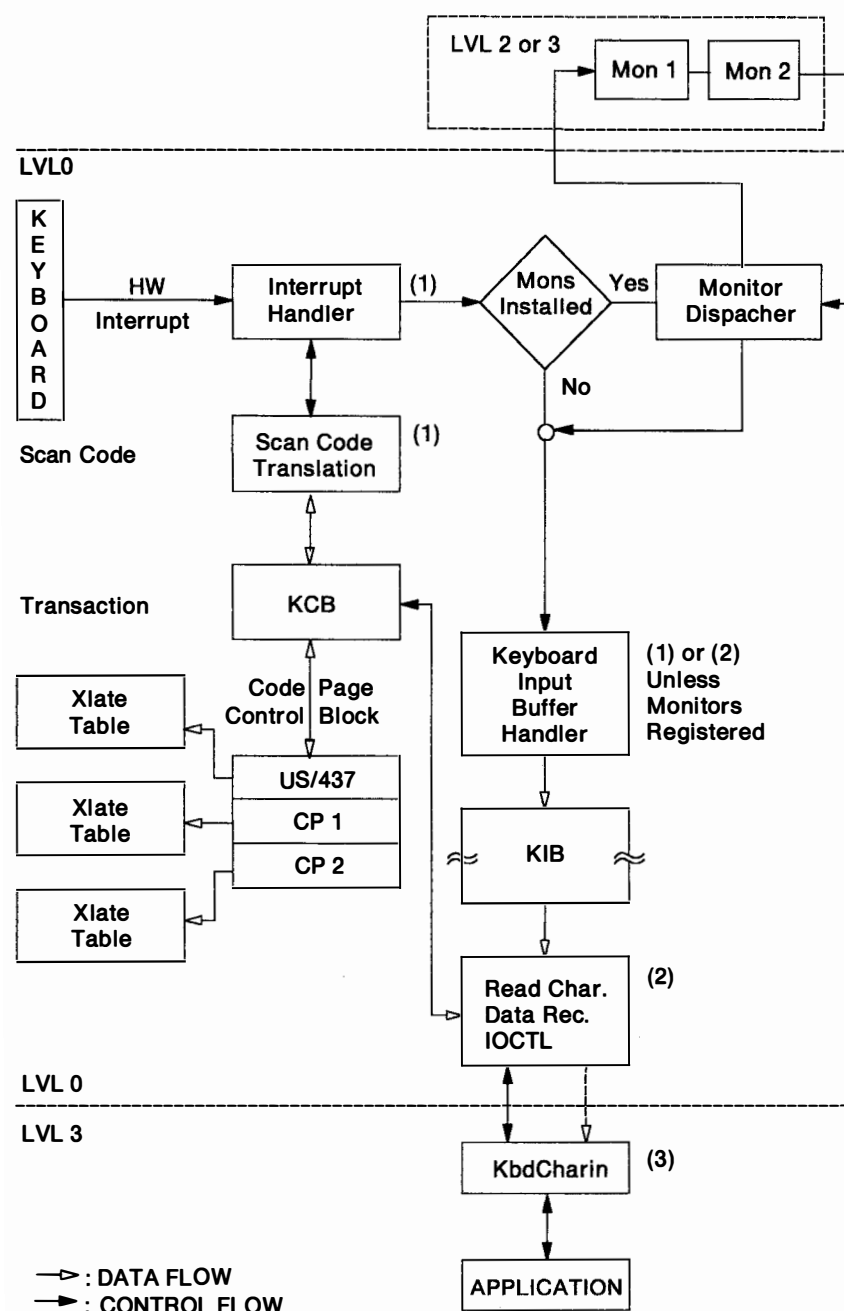


Figure 11-1. Keyboard System Structure

The physical Keyboard device driver interfaces the physical keyboard to applications through various levels of routines (see Figure 11-1) where:

- (1) is part of the physical device driver interrupt handler.
- (2) is part of the physical device driver strategy routine.
- (3) is part of the base subsystem/router.

Keyboard Initialization

At the end of CONFIG.SYS processing (after the CODEPAGE and DEVINFO statements have been processed), the code pages are initialized by calling KbdSetCp.

Keyboard Run Time Operation

When a session is started, a default logical keyboard already exists. This keyboard is identified to the subsystem by a *zero handle*. Any program can share the keyboard by using Handle 0. If multiple programs use the default keyboard, they must coordinate their access to it. The default keyboard logically terminates when the session terminates.

If a program wants a logical keyboard separate from the default logical keyboard, it does a KbdOpen. This open creates a new logical keyboard, but does not make the physical to logical bond. As a result of an open, a handle unique within a process is returned to the caller, which is later used to identify the logical keyboard. The handle ownership is tied to the process; handles are not inherited. Use of KbdOpen does not prohibit the process from using the default keyboard.

To make the physical to logical bond, the process issues the KbdGetFocus, using the handle identifying the logical keyboard. Once the bond is made, the logical keyboard can receive keystrokes. Note that type-ahead keystrokes are not possible before the bond is made. The Keyboard API can only be used when the bond is made, or with Handle 0, when no other handle has the bond.

The bond represents a foreground keyboard; one exists for each session. This is either the default or a created logical keyboard. Breaking the bond is done with the KbdFreeFocus call. If other threads have a KbdGetFocus outstanding, the thread having the highest priority gets the bond. If there are no KbdGetFocus calls outstanding, the physical keyboard reverts to the default keyboard.

A logical keyboard is destroyed with a KbdClose. The close does a free focus, flush buffer, and deallocates the KCB and related memory. The close is done by the process kill mechanism, if not done by the program.

Keystroke Monitors

Some applications need to view the raw keystrokes as they arrive from the keyboard at interrupt time. These applications might consume, modify, or replace keystrokes. This is made possible by the keystroke monitor function.

An application operating with the OS/2 system cannot register a keyboard monitor for a Presentation Manager Session. When DosMonReg is issued with keyboard devices, the INDEX parameter indicates which OS/2 session, requested by the caller, is to register the monitor. Notice that the INDEX parameter indicates an OS/2 session from 0 – 15 (see DosGetInfoSeg).

An INDEX parameter value of -1 indicates that the session of the calling thread is being requested. If the caller requests either the Presentation Manager session or the DOS Session, a MONITORS_NOT_SUPPORTED error is returned. For both of these sessions, the error return code is generated, regardless of the caller's specific usage of the INDEX parameter (explicit/implicit).

The size of the physical Keyboard device driver's monitor chain buffer is 16 bytes. This is the value to be used in calculating the sizes of the input/output buffers required for the DosMonReg function. The physical Keyboard device driver supports device monitors. This physical device driver passes its information to the monitors in packets. Figure 11-2 on page 11-4 describes the content and format of keystroke monitor packets.

MonFlagWord:		Word
Character Data Record	XlatedChar:	Byte
	XlatedScan:	Byte
	DBCS Status:	Byte
	DBCS Shift	Byte
	Shift State:	Word
Record	Milliseconds:	DWord
KbdDDFlagWord		Word

Figure 11-2. Keystroke Monitor Data Packet Definition

MonFlagWord (Lower Byte): Monitor Dispatcher Flags.

- Bits 7-3** RESERVED. Should be passed untouched for packets that are passed on. Should be set to 0 on packets that are inserted by a monitor.
- Bit 2** FLUSH. This is a flush packet. No other information in the packet has meaning. Monitor should flush its internal buffers and pass the packet quickly.
- Bit 1** CLOSE. Not used by keystroke monitors.
- Bit 0** OPEN. Not used by keystroke monitors.

MonFlagWord (Upper Byte): Original Scan code, as read from the hardware. If 0, this packet was inserted for other reasons, see "KbdDDFlagWord." Monitors pass this field untouched or put 0 here if they insert a packet.

CharData Record: See the IOCTL "Function 74H — Read Character Data Records" on page 18-93 for information on this field.

KbdDDFlagWord: Has the following bits:

- Bits 15-14** Available. These bits are available for communication between monitors; they are not used by the physical device driver. The monitor applications coordinate the use of these flags.
- Bits 13-10** RESERVED=0. Monitors should pass these flags as is. They should set these flags to 0 in packets they create.
- Bit 9** ACCENTED. This key is translated using the previous key passed, which is an accent key. See 10H ACCENT KEY on page 11-6. Where an accent key is pressed, and the following key doesn't use the accent, a packet containing the accent character itself is first passed with this bit set. The scan code field of MonFlagWord (see above) would be 0, indicating a non-key generated record. A valid packet containing that following keystroke is then passed without this bit set.
- Bit 8** MULTIMAKE. The translation process sees this scan code as a typematic repeat of a toggle key or a shift key. Because toggle and shift keys only change state on the first make after each key-break, no state information is changed. For example, the NumLock toggle bit in the shift status WORD is not changed, even though this can be the NumLock key. If this key is a valid character, it does not go into the KIB once this bit is set.

- Bit 7** SECONDARY. The scan code prior to the one in this packet was the Secondary Key Prefix (see below).
- Bit 6** KEY BREAK. This record is generated by the release (the break) of the key involved.
- Bits 5-0** KEY TYPE. Numeric field that tells the physical device driver that this is a key that requires action. The number in this field is filled in during the translation of the scan code. The value here allows the driver to act on keystrokes without regard for what scan codes the keyboard uses or character codes that the current translation process may be using. The following values are currently defined:
- Value for keys that are always placed in the KIB. Zero = no special action, always place in KIB.
 - Values acted on prior to passing packet to monitors: Except for the final keystroke of the DUMP key sequences, all of these values are passed on to the monitors. They are not placed in the KIB. The XlatedChar and XlatedScan fields are undefined for these values:
- 01H** ACK. This scan code is a keyboard acknowledge. IBM Personal Computer AT* attached keyboards set this value on a FAH scan code.
- 02H** SECONDARY KEY PREFIX. This scan code is a prefix scan code generated by the Enhanced Keyboard and indicating the next scan code coming is one of the secondary keys that exists on that keyboard. Usually set on a E0H scan code or a hex E1 scan code.
- 03H** KBD OVERRUN. This scan code is an over-run indication from the keyboard. On an IBM Personal Computer AT attached keyboard, this value would be set on a FFH scan code.
- 04H** RESEND. This scan code is a *resend* request from the keyboard. On an IBM Personal Computer AT attached keyboard, this value would be set on a FEH scan code.
- 05H** REBOOT KEY. This scan code completes the multi-key restart sequence. On an IBM Personal Computer AT attached keyboard, this value would be used when the Ctrl + Alt + Delete sequence is used.
- 06H** DUMP KEY. This scan code is completes the multi-key Stand Alone Dump request sequence. On an IBM Personal Computer AT attached keyboard, this value would be used on completion of the second consecutive press of Ctrl + Alt + NumLock without other keystrokes between the two presses.
- 07H – 0AH** See entries below.
- 0BH** INVALID ACCENT COMBINATION. This scan code follows an accent scan code but the combination is not valid, and neither key is put in the KIB.
- Note:** This is set, if the Canadian-French code pages are in use.
- 0CH** SYSTEM-DEFINED HOT KEYS.
- 0D – 0FH** RESERVED. Treated as undefined. See entry 3FH.
- Values acted on after passing packet to monitors: Except where noted, these are placed in the KIB when the physical device driver is in Binary mode but are not placed in the KIB when the physical device driver is in ASCII mode. (Also listed are those that never get placed in the KIB.)

* Trademark of the IBM Corporation

- 07H** SHIFT KEY. This scan code translates as a shift key and affects the shift status fields of the CharData record, but does not generate a defined character so it is not placed in the KIB. The XlatedChar field is undefined. The scan code field is 0.
- 08H** PAUSE KEY. This scan code is translated as the key sequence meaning *pause*. On an IBM Personal Computer AT attached keyboard, this value is used when the Ctrl + NumLock sequence is used. The key itself is not placed in the KIB.
- 09H** PSEUDO-PAUSE KEY. This scan code is translated into the value that is treated as the Pause key when the physical device driver is in ASCII mode. On most keyboards, this would be when the Ctrl + S combination is used. The key itself is not placed in the KIB.
- 0AH** WAKE-UP KEY. This scan code follows a Pause key or Pseudo-Pause key, which causes the pause state to be ended. The key itself is not placed in the KIB.
- 10H** ACCENT KEY. This scan code is translated, and used as a key to alter the translation of the next key to come in. The packet containing this value is passed when the accent key is hit, but it is not put into the KIB, unless the Accented bit is *on*. (See ACCENTED bit on page 11-4.) The next key determines this decision. If the next key is one that can be accented, then it is passed by itself with the Accented bit *on*. If that next key cannot be accented by this accent, then two packets are passed. The first contains the character to print for the accent itself, has the Accent key value, and the Accented flag (which allows the packet to be put in the KIB). The second packet contains a regular translation of that following key.
- Note:** The two packets get passed for every language except Canadian-French. See entry 0BH.
- 11H** BREAK KEY. This scan code is translated as the key sequence meaning *break*. On the IBM Personal Computer AT attached keyboard, this value is used where the Ctrl + Break sequence is used.
- 12H** PSEUDO-BREAK KEY. This scan code is translated into the value that is treated as the Break key when the physical device driver is in ASCII mode. On most keyboards, this would be when the Ctrl + C combination is used. Notice that the event generated by this key is separate from that generated by the Break key when in the binary mode.
- 13H** PRINT SCREEN KEY. This scan code is translated as the key sequence meaning *Print Screen*. On an IBM Personal Computer AT attached keyboard, this value is used where the Shift-PrtSc sequence is used.
- 14H** PRINT ECHO KEY. This scan code is translated as the key sequence meaning *Print Echo*. This value is used where the Ctrl + PrtSc sequence is used.
- 15H** PSEUDO-PRINT ECHO KEY. This scan code is translated into the value that is treated as the Print Echo key, when the physical device driver is in ASCII mode. On most keyboards this would show as the Ctrl + P combination.
- 16H** PRINT-FLUSH KEY. This scan code is translated into the key sequence Print-Flush. This value is used where the Ctrl + Alt + PrtSc sequence is used.
- 17H – 2FH** RESERVED=0. Treated as undefined. See entry 3FH.
- Values for packets not generated by a keystroke:
- 30H – 37H** RESERVED
- 38H – 3EH** RESERVED. Treated as undefined. See entry 3FH.

- Value for keys the translation process does not recognize:

3FH **UNDEFINED.** This scan code, or its combination with the current shift state, is not recognized in the translation process.

Special Key Processing

The operating system examines each incoming keyboard character to determine if it should cause an asynchronous signal to be sent to some process; for example, Ctrl + C. The physical Keyboard device driver responds to the keystrokes as follows:

<i>Table 11-1. Special Key Processing</i>			
Keys	Mode	Event Signalled or Signal Handler Called	Value Placed in KIB
Ctrl + C	Binary	Nothing	03H
Ctrl + C	ASCII	SIGINTR	Nothing
Ctrl + S	Binary	Nothing	13H
Ctrl + S	ASCII	Event_CtrlScrLk	Nothing
Ctrl + P	Binary	Nothing	10H
Ctrl + P	ASCII	Event_CtrlPrtSc	Nothing
Ctrl + Break	Binary	SIGBREAK	00:00H
Ctrl + Break	ASCII	SIGINTR	Nothing
Ctrl + NumLock	Binary	Event_CtrlScrLk	Nothing
Ctrl + NumLock	ASCII	Event_CtrlScrLk	Nothing
Ctrl + PrtSc	Binary	Event_CtrlPrtSc	Nothing
Ctrl + PrtSc	ASCII	Event_CtrlPrtSc	Nothing
PrtSc	Binary	Event_ShftPrtSc	Nothing
PrtSc	ASCII	Event_ShftPrtSc	Nothing

Notes:

1. The value *xxH* represents the translated ASCII field in the character data record.
2. The value *xx:yyH* represents the translated ASCII, that is, translated scan code fields of the character data record.
3. Ctrl + NumLock has no effect on an IBM Enhanced keyboard; the Pause key provides this function.

Key	Meaning
Hot Key	Change sessions
Ctrl + Alt + Del	System IPL (restart)
Ctrl + Alt + NumLock	Pressed twice means system dump
PrtScr	Print the current display screen. (Print Screen on an IBM Enhanced keyboard)

In addition to passing individual characters, the physical device driver must specifically recognize the character (or character sequence, or other special action) that indicates the special signal, or hot key, to the Session Manager. When this event is recognized, the SendEvent Device Helper service is invoked.

Notice that the System Hot Key is defined by IOCTL Category 4 "Function 56H — Set Session Manager Hot Key" on page 18-81.

Chapter 12. Physical Mouse Device Driver

Two classes of pointing devices are supported, relative and absolute. A *relative pointing device* reports relative motion, that is, how far it has moved. An example of a relative pointing device is a mouse, or a track ball. An *absolute pointing device* reports absolute positions within some predefined work space; there is no real concept of relative motion. An example of an absolute pointing device is a touch-sensitive screen.

Listed below are commonly used terms and their definitions. These terms are frequently used throughout this chapter:

- **MOUSE\$.** The OS/2 system-provided pointing device driver name, which is defined in the device header field of MOUSE.SYS.
- **IDC.** Inter-Device Communication.
- **Device-Independent Device Driver.** Another way of referring to MOUSE.SYS, which handles all the IDC interfaces defined on the following pages.
- **Device-Dependent Device Driver.** Hardware-specific device driver that communicates with MOUSE.SYS through the IDC for additional pointing device support. The OS/2 operating system provides pointing device support for the following:

IBM Mouse

IBM 8516 Touch Display

Microsoft[™] Mouse

Logitech[™] Mouse

PC Mouse[™] Systems Mouse

Any pointing device that is compatible with the above devices will be supported.

Generic Pointing Device Support

The OS/2 operating system provides a physical Mouse device driver called MOUSE.SYS that tries to detect the type of pointing device currently installed on the system. Once it detects the existence of a particular pointing device, it dynamically sets up support for that device.

If the physical Mouse device driver is unable to detect the presence of a pointing device, the install program prompts the user for pointing device information. The install program then sets the appropriate statements for pointing device support in the CONFIG.SYS file. The physical Mouse driver will set up to support the first pointing device that it finds.

High-Level Design

During device driver initialization time, the physical Mouse device driver first checks to see if the TYPE = override has been used. If the DEVICE = C:\OS2\MOUSE.SYS line in CONFIG.SYS contains a TYPE = override, then pointing device support is established through an IDC interface with the device-dependent device driver name following TYPE = . The device-dependent device driver must be loaded before MOUSE.SYS.

[™] Microsoft is a trademark of the Microsoft Corporation

[™] Logitech is a trademark of Logitech, Inc.

[™] PC Mouse is a trademark of the Metagraphics/Mouse Systems

If a TYPE = override has not been specified, it is assumed that generic pointing device support is desired. The generic pointing device driver detects if the system is a Family1 (non-ABIOS) system, or a Family2 (ABIOS) system.

Application Level Interface

An IOCTL to get the device type is provided. The install program calls the IOCTL sometime during the install process to find out if a pointing device has been found. If no pointing device is found, the install program prompts the user for the pointing device that is installed. The install program can also give the user a chance to override the pointing device found by MOUSE.SYS.

Note: A situation where the user would override the pointing device support setup by the generic pointing device driver is unlikely. (Multiple pointing devices would be necessary in order for this situation to occur.)

In some cases, the generic pointing device driver might not be able to detect the pointing device, even though it is plugged into the system. The install program prompts the user for the pointing device type and updates the CONFIG.SYS file with a TYPE = override, if a hardware-dependent device driver is currently provided for the specified device.

Physical Mouse Device Driver Considerations

System Install ensures that physical Mouse device driver initialization takes place prior to physical ASYNC device driver initialization. This allows the physical ASYNC device driver to determine that it is not responsible for servicing that port, which ensures that physical Mouse device drivers are not preempted from the COMx ports by the physical ASYNC device drivers.

Note: When manually changing CONFIG.SYS, the user must place the mouse DEVICE = statements before ASYNC DEVICE = statements.

Adding Support for a Unique Pointing Device

OS/2 2.0 provides a method for supporting additional pointing devices. Pointing device support can be obtained by writing a device-dependent device driver for the device. This physical device driver will communicate directly with the OS/2-provided device driver MOUSE.SYS, that is, the device-independent device driver, through the IDC interfaces that follow.

MOUSE.SYS IDC Interface

The IDC provided by MOUSE.SYS support the following:

- Process_Packet
- Disable_Support
- Process_Absolute
- Open_Mouse
- Close_Mouse
- Query_Activity

Process_Packet: This request is issued by the device-dependent device driver to pass a relative-pointing-device event to MOUSE.SYS. Process_Packet is non-reentrant. It is the responsibility of the device-dependent device driver to ensure that calls to Process_Packet are never nested. Process_Packet should be issued only when the pointing device is enabled.

Input:

AX = Process_Packet function code, 0001H.

Common-event buffer initialized. The offset of the common-event buffer was passed on the Read_Enable request. This data buffer is used to pass both relative and absolute pointing-device events. When the physical device driver issues a Process_Packet request, the buffer has the following format:

Event_Data Struc

Event	dw	?	; Description of event
Col_Motion	dw	?	; Relative column motion
Row_Motion	dw	?	; Relative row motion

Event_Data Ends

Before the physical device driver makes the request, initialize the buffer. Notice that positive motion is *up* and to the right. *Up* can also be viewed as moving a mouse device away from the user. Negative motion is the opposite of positive.

Output: No specific output for request.

Disable_Support: This request is used by the device-dependent device driver to inform MOUSE\$ that it has *deinstalled*. When the request is received, MOUSE\$ enters a disabled state. It returns the error, DEVICE NOT READY, for subsequent requests from the system.

Note: Do not issue this request under normal circumstances.

Input:

AX = Disable_Support function code, 0002H.

Output: No specific output for the request.

Process_Absolute: This request is issued by the device-dependent device driver to pass an absolute-pointing device event, usually from a touch-sensitive screen, to MOUSE.SYS. Process_Absolute is non-reentrant. It is the responsibility of the device-dependent device driver to ensure that calls to Process_Absolute are never nested. Process_Absolute should be issued only when the pointing device is enabled. The reported event is mapped into the current display mode and appears to the system to be a mouse event.

Input:

AX = Process_Absolute function code, 0003H.

Common-event buffer initialized. The offset of the common-event buffer was passed on the Read_Enable request. This data buffer is used to pass both relative and absolute pointing-device events. When issuing a Process_Absolute request, the buffer has the following format:

Event_Data Struc

Event	dw	?	; Description of event
Row_Pos	dw	?	; Row position of event
Col_Pos	dw	?	; Col position of event
Row_Size	dw	?	; Number of row units
Col_Size	dw	?	; Number of column units

Event_Data Ends

MOUSE\$ uses Row_Size and Col_Size to map the absolute event into the current display mode. These values should reflect the maximum allowed position that could be reported. All values are zero-based. The *event* field should never indicate that motion was associated with

mouse PDD

the event. MOUSE\$ determines if motion occurred. The upper-left corner of the device should be reported as location 0,0.

Initialize the buffer before making the request.

Output: No specific output for the request.

Open_Mouse: This function is invoked by another Input Device Driver to get an Mouse IDC Interface handle. This handle is a required parameter for the Close_Mouse and Query_Activity functions. This function supports a maximum of five open Mouse IDC handles at any given point in time. Attempts to open a 6th Mouse IDC handle are returned with an NO_HANDLES_AVAILABLE error code.

Input:

AX = Open_Mouse function request code, 0004H
DS = MOUSE\$ DD DS value
ES = Calling DD DS value
BX = Undefined
CX = Undefined
DX = Undefined
SI = Undefined
DI = Undefined

Output:

AX = Error Return code if Carry is set; undefined if Carry is clear
BX = Mouse IDC handle
DS = MOUSE\$ DD DS value
ES = Calling DD DS value
CX = Undefined
DX = Undefined
SI = Undefined
DI = Undefined

Error Return Codes:

NO_HANDLES_AVAILABLE

Remarks: This function is valid any time after device driver initialization.

Close_Mouse: This function is invoked by another physical device driver to free a Mouse IDC Interface handle. The input handle value must have been generated using the Mouse IDC Function 0004H, Open_Mouse. If the input Mouse IDC handle is invalid, then an INVALID_HANDLE error code is returned.

Input:

AX = Close_Mouse function request code, 0005H
BX = Mouse IDC handle
DS = MOUSE\$ DD DS value
ES = Calling DD DS value
CX = Undefined
DX = Undefined
SI = Undefined
DI = Undefined

Output:

AX = Error Return code if Carry is set; undefined if Carry is clear
DS = MOUSE\$ DD DS value
ES = Calling DD DS value
BX = Undefined
CX = Undefined

DX = Undefined
SI = Undefined
DI = Undefined

Error Return Codes:

INVALID_HANDLE

Remarks: This function is valid any time after device driver initialization.

Query_Activity: This function is invoked by another physical device driver to get the current Mouse Activity Status, in relation to the last time this function was called, or since the caller's handle was opened. This function does not provide the caller with the current Mouse State only the data from a last call perspective. A return value of 0 in the BX register (without an error condition) indicates that no mouse device activity has been recorded since the last call. In addition, the return status is on a system wide basis, that is, there is no evaluation for a particular session's activity. If the input Mouse IDC handle is invalid, then an INVALID_HANDLE error code is returned.

Input:

AX = Query_Activity function request code, 0006H
BX = Mouse IDC handle
DS = MOUSE\$ DD DS value
ES = Calling DD DS value
CX = Undefined
DX = Undefined
SI = Undefined
DI = Undefined

Output:

AX = Error Return code if Carry is set; undefined if Carry is clear
BX = Mouse Activity Status
DS = MOUSE\$ DD DS value
ES = Calling DD DS value
CX = Undefined
DX = Undefined
SI = Undefined
DI = Undefined

The Mouse Activity Status is a returned bit mask. A set bit is defined as having a value of 1. A set bit indicates the description status or event occurred. The bit definitions for the Mouse Activity Status parameter are as follows:

Bits 15-11	Reserved=0
Bit 10	Button 5 usage, without motion
Bit 9	Button 5 usage, with motion
Bit 8	Button 4 usage, without motion
Bit 7	Button 4 usage, with motion
Bit 6	Button 3 usage, without motion
Bit 5	Button 3 usage, with motion
Bit 4	Button 2 usage, without motion
Bit 3	Button 2 usage, with motion
Bit 2	Button 1 usage, without motion
Bit 1	Button 1 usage, with motion
Bit 0	Pure motion was recorded.

Note: The Mouse Device-dependent IDC Interface allows a device with up to five buttons to be supported. The OS/2 operating system provides specific device support for only two and three button mice.

Error Return Codes:

INVALID_HANDLE
UNKNOWN_COMMAND

Remarks This function is valid only when OS/2 AIM Support is enabled. When Special Needs Support is Inactive (AIM_Active = FALSE), IDC Query_Activity function requests are not allowed, and are returned to the caller with an UNKNOWN_COMMAND error code. For details, see the Category 11 "Function 41H – System Notifications for Physical Device Drivers" on page 18-173 for information on OS/2 AIM Support for physical device drivers.

Device-Dependent Device Driver IDC Interface

The IDC provided by the device-dependent device driver must support the following requests:

- Query_Config
- Read_Enable
- Read_Disable
- Disable_Device
- Enable_Device

Query_Config: This request is issued by MOUSE\$ and determines the operating characteristics of the attached hardware. The format follows:

Input:

AX = Query_Config function code, 0001H.
DI = Offset in mouse data segment of where to write the configuration data.
ES = Must be loaded with the MOUSE\$ data segment/selector prior to the request. The configuration data has the following format:

Config_Data Struc

```

Length      dw      ? ; Total length in bytes (7)
Num_Mics    db      ? ; Device resolution in mickeys/centimeter
Num_Butt    db      ? ; Number of buttons (up to 7 supported)
Dev_IRQ     db      ? ; Device IRQ level
Mouse_Type  db      ? ; Type of mouse attached:
                ;      0 = Unknown
                ;      1 = Bus mouse
                ;      2 = Serial mouse
                ;      3 = InPort mouse
                ;      4 = Pointing Device Port Mouse
                ;      5 = IBM 8516 Touch Display
                ;      6-255 = Reserved
Com_Num     db      ? ; Com port number of the attached pointing device
                ; 0 - If non-serial device

```

Config_Data Ends

Initialize the Length field to 7 before issuing the request. The Num_Mics field for an absolute device should reflect some reasonable measure of the number of device units per centimeter.

Output: Configuration data structure filled if no errors.

Read_Enable: This request serves two purposes. It gives the offset of the common-event buffer in the MOUSE\$ data segment. (This offset should be saved.) It also informs the device-dependent device driver that it can start sending event data to MOUSE\$, using the Process_Packet and Process_Absolute requests. The device-dependent device driver also should attach itself to MOUSE\$; using the Device Helper service, AttachDD, when it receives this request.

Input:

AX = Read_Enable function code, 0002H.

DI = Offset to common-event buffer. This buffer resides in the MOUSE\$ data segment. To address it, use the MOUSE\$ data segment/selector value (depending on the processor mode) returned on the AttachDD call. The common-event buffer has enough space to format either a relative or absolute event. For the two common-event buffer formats, see "Process_Packet" on page 12-2, and "Process_Absolute" on page 12-3.

Output: No specific output for the request.

Read_Disable: This request is issued by MOUSE\$ when it no longer accepts Process_Packet or Process_Absolute requests. References to the common-event buffer should be discontinued. However, MOUSE\$ can enable the interface again by sending another Read_Enable request. Normally this request is not issued.

Input:

AX = Read_Disable function code, 0003H

Output: No specific output for the request.

Disable_Device: This request is used by MOUSE\$ when placing a pointing device in a disabled state. Upon completion of this request, no Process_Packet or Process_Absolute requests are to be made to MOUSE\$ until an Enable_Device request is received. There is no requirement that the hardware itself be disabled; only that no Process_Packet or Process_Absolute requests be issued. Receipt of a Disable_Device request is not considered an error if the pointing device is already disabled.

Input:

AX = Disable_Device function code, 0005H.

Output: No specific output for the request.

Enable_Device: This request is used by MOUSE\$ when resetting a pointing device to an enabled state. Process_Packet and Process_Absolute requests are made to MOUSE\$ when a pointing device is in an enabled state. Receipt of an Enable_Device request is not considered an error, if the pointing device is already enabled.

Input:

AX = Enable_Device function code, 0004H.

Output: No specific output for the request.

Chapter 13. Physical Parallel Port Device Driver

The OS/2 physical Parallel Port device driver is a base physical device driver, which supports the parallel port device interface for OS/2 2.0. There are two distinct versions of the physical Parallel Port device driver:

- PRINT01.SYS** Supporting parallel port adapters on Family One bus machines
- PRINT02.SYS** Supporting parallel port adapters on Family Two machines (machines containing Micro Channel architecture).

The Family One physical Parallel Port device driver manipulates the hardware ports through I/O instructions. The Family Two physical Parallel Port device driver does not access the hardware ports. Advanced BIOS (ABIOS), which executes on Family Two machines, replaces the hardware port manipulation done on the Family One machines.

Only one of these physical device drivers will be resident for each machine running in OS/2 2.0. Since the physical Parallel Port device driver is a base physical device driver, OS/2 2.0 determines the system's hardware configuration and automatically loads the appropriate physical device driver when the system is initialized. A `DEVICE=` command in `CONFIG.SYS` is not required to install the physical Parallel Port device driver.

Overview

The physical Parallel Port device drivers service parallel port requests from DOS and OS/2 applications (including those running under the Presentation Manager). Requests are serialized so that mixed printer output will not occur. Printer requests include:

- Initializing a parallel port device
- Sending characters to a parallel port device
- Returning the status of a parallel port device
- Handling IOCTL function calls, including the setting of the parallel port device's frame control (characters per line and lines per inch).

In addition, the physical Parallel Port device drivers include character device monitor support. The physical Parallel Port device driver sends data to a character device monitor, if one is installed.

The primary method of communicating with the Parallel Port device driver is through a request packet. When the Parallel Port device driver receives a request packet, it determines which device the request is for. If a request is not already in progress, the physical device driver calls an appropriate routine to handle the request. If a request is in progress, the physical device driver blocks the caller's thread until the in-progress request has completed. Once the in-progress request has been serviced, the physical device driver will run the caller's thread and process the waiting request.

To make a request of the printer in OS/2 2.0, an application pushes parameters (for example, the address of characters to be written) onto its stack, and calls the file system API (for example, `DosWrite`). The file system:

- Determines that the request is for the Parallel Port device driver (based on the specified device handle)
- Creates a request packet
- Issues a call to the Parallel Port device driver strategy routine.

To make a request of the printer in the DOS Session, an application moves the parameters into predefined registers, and issues an `INT 17H` or `INT 21H`. The `INT 17H` is intercepted by the virtual Parallel Port device driver. Refer to *OS/2 2.0 Virtual Device Driver Reference* for more information. The `INT 21H` request is converted into a request packet and issued to the device driver strategy routine.

Replacing the Parallel Port Device Driver

The Parallel Port device drivers, both the Family One and Family Two, contain four device headers in their data segments with the reserved device names PRN, LPT1, LPT2, and LPT3 corresponding to the three physical devices supported by this physical device driver. Notice that the names PRN and LPT1, although logically appearing different, correspond to the same physical device.

The parallel port device driver is replaceable, that is, it is deinstalled by:

- Creating another physical Parallel Port device driver with the device name in the device header identical as one listed above; then
- Specifying the new physical Parallel Port device driver in a `DEVICE =` statement in `CONFIG.SYS`.

Note: Installation of the files associated with the new device driver, and the addition of the corresponding `DEVICE =` statement in `CONFIG.SYS`, is accomplished through the `DDINSTALL` utility, and the Device Driver Profile (DDP) for the new device driver.

When the `DEVICE =` statement for the new Parallel Port device driver is encountered in `CONFIG.SYS` during system initialization, the system generates a `DEINSTALL` request packet, and sends it to the Parallel Port device driver. The OS/2 physical Parallel Port device driver receives the `DEINSTALL` request packet, and releases its resources. See "14H / `DEINSTALL`" on page 15-20 for more information.

If the physical Parallel Port device driver successfully releases its resources, it returns a successful completion on the Deinstallation request. When the physical Parallel Port device driver has been deinstalled, the system then generates an `INSTALL` request packet, and sends it to the new device driver. See "0H / `INIT`" on page 15-5 for more information.

Parallel Port IRQ Performance

The Programmable Interrupt Controller (PIC) has been rotated under OS/2 2.0, giving IRQ3 (serial port) the highest priority in the system. This has given IRQ7 (parallel port) a much higher priority than it would otherwise have. It is now higher in priority than the keyboard or the mouse. A system unit running multiple parallel port devices, which can receive data quickly on a combination of serial/parallel port devices at a high rate of speed (for example, a laser printer with several megabytes of memory), can experience slow user input performance from the mouse and keyboard.

Several alternatives exist to improve user input performance. One alternative, for attended print servers, is to ensure the system unit supports a DMA parallel port and/or a DMA serial port. The physical Parallel Port device driver takes advantage of direct memory access (DMA), which reduces the number of hardware interrupts in the system. Fewer high priority interrupts in the system allow the processor to spend more time processing lower priority interrupts.

Another alternative is to convert a parallel port device to run using the serial port, and to reduce the bit rate to the serial device. However, the ASYNC adapter must support extended hardware buffering. For more information on this approach, refer to Chapter 8, "Physical ASYNC (RS232-C) Communications Device Driver" on page 8-1.

A last alternative for those situations where user responsiveness is extremely important, is to reduce the number of print files being sent to the print server by moving some of the devices to other system units (thus balancing the workload among several system units available), or to reduce the user installable memory within the device.

DMA Parallel Port Support

The OS/2 2.0 physical Parallel Port device driver for PS/2 systems supports DMA parallel ports using the BIOS parallel port functions found on those system units.

Using the parallel port in a Programmed I/O (PIO) methodology, causes the processor to be involved on each byte transmitted. This prevents the processor from performing other operations.

Using the parallel port in a DMA (Direct Memory Access) methodology, causes the processor to be involved on each buffer transmitted. The larger the buffer, the fewer interrupt to the CPU, and the more time the CPU can spend doing other useful work.

Code Page Support

The physical Parallel Port device driver provides code page support only through the OS/2 character device monitor mechanism (for example, the spooler). In addition, three printer IOCTL commands support code page and font switching so that an application has the ability to change the active code page and font in the middle of its print job.

Parallel Port (Printer) Monitors: The physical Parallel Port device driver defines a data stream for each physical Parallel Port device. In addition, each printer data stream can be monitored by parallel port device monitor applications, registered with one of two monitor chains. Character device monitor support is provided by the Parallel Port device driver, therefore, for each physical Parallel Port device.

To register a monitor for one of the Parallel Port devices, an application issues the DosMonOpen and DosMonReg monitor function calls. DosMonReg generates an IOCTL request, for the application, to the physical Parallel Port device driver to register a monitor on one of the two monitor chains associated with the device. See Category 10 "Function 40H – Register Monitor" on page 18-168. When the physical Parallel Port device driver receives this request, it:

- Creates a monitor chain for that device by calling the DevHlp, MonitorCreate, if none was previously created as the result of another register monitor IOCTL request
- Registers the monitor with the monitor chain by calling the DevHlp, Register.

By setting the DosMonReg index parameter appropriately, the application specifies, on which of the two monitor chains, its monitors are registered. The first monitor chain (specified by the application by setting Index=1), is considered to be the data chain, and is used by the physical Parallel Port device driver to send data to a monitor. The second monitor chain (specified by the application by setting Index=2), is considered to be the code page chain.

To insure that printer requests are processed in order, the physical Parallel Port device driver places all data and code page requests into only the first monitor chain (referred to as the data chain). That is, the Parallel Port device driver issues MonWrite calls only to the data chain. The physical Parallel Port device driver does not place data into the code page chain by calling MonWrite.

To improve performance, the second monitor chain (code page chain) is used by the Parallel Port device driver primarily to receive the results of a code page request. Processes that issue code page requests are blocked until they receive an indication that their request is valid. Since the number of code page requests is negligible compared to the number of data requests, parallel port monitors can respond more quickly and efficiently to the physical device driver through the code page chain.

In summary, parallel port monitors:

- Receive all data and code page requests from the first monitor chain (see DosMonRead)
- Return data requests to the first monitor chain (see DosMonWrite)

- Return code page requests to the first or the second monitor chain (see DosMonWrite), depending on the position of the monitor when other monitors are registered with the chain (see below).

Note: The physical Parallel Port device driver expects to receive the monitor responses to code page requests on the code page chain. If monitor responses to code page requests are not received from any monitor on the code page chain, unpredictable results can occur.

The spooler is an example of an application that registers a parallel port monitor with both the data chain and the code page chain for a parallel port device. Because no requests are placed into the second monitor chain by the Parallel Port device driver, the monitor input buffer specified, when the spooler registers the monitor with the second chain (see DosMonReg, with Index=2), is not used.

Parallel port monitors must be designed with extreme care. They must be positioned carefully when other parallel port monitors, in particular the spooler, are registered:

- If an application monitor wishes only to process character data, it registers a parallel port monitor, only on the data chain (Index = 1) when:
 1. It is the only monitor in the monitor chain (for example, the spooler is not loaded).
 2. It is registered in a position to process the data after the spooler. This monitor never sees code page requests from the spooler, because the spooler automatically sends these requests back to the physical device driver through the code page chain.
- If an application wishes to process both character data and code page requests, it registers a parallel port monitor on the data chain (Index = 1), and a parallel port monitor on the code page chain (Index = 2), when:
 1. It is the only monitor in the monitor chain. It must expect to receive both data and code page requests on the data chain. It must respond to the code page requests on the code page chain as quickly as possible.
 2. It is registered in a position to process data after the spooler. It must expect to receive character data from the spooler on the data chain, and the code page requests from the spooler on the code page chain.

Because the spooler passes the code page results along the code page chain before all the data has been spooled and released to be printed, a parallel port monitor cannot easily synchronize the code page requests with the data requests.

- If a parallel port monitor wishes to process character data and code page requests, and it is registered in a position to process data before the spooler, it registers on the data chain (Index = 1) only.

All data and code page requests are sent to the first (non-spooler) monitor on the data chain. Because the spooler receives data and code page requests from the data chain only, this monitor must pass on to the spooler through the data chain, all the information it receives in the order that it is received.

IOctl Support of Code Page and Font Switching: The Parallel Port device driver has three printer-specific IOctl commands to support the code page and font switching provided in OS/2 2.0. In order to support these IOctl calls, there are Font Monitor Buffer command/responses in the monitor interface between the physical Parallel Port device driver and the spooler.

All of the actual code page and font switching functions for printers are provided by the code page switcher, or Presentation Manager device drivers. When the spooler is started, it checks to see if code page support is required. If it is, the spooler causes the code page switcher to be loaded and initialized. The spooler interfaces with the code page switcher through the DosPFSxxx API functions. If a DEVINFO= statement is not specified in CONFIG.SYS, the spooler redirects code page requests to the Presentation Manager device drivers.

The following illustrates the dialog between the physical Parallel Port device driver and the spooler, when the physical Parallel Port device driver receives one of these special IOCTL requests from an application:

1. The physical Parallel Port device driver receives an IOCTL request from an application to activate a code page.
2. If the spooler is loaded, the physical Parallel Port device driver sends a Font Monitor Buffer command to the spooler in the form of a special monitor record placed into the data monitor chain (see "MonWrite" on page 17-48).
3. The spooler receives this special monitor record from its monitor input buffer registered with the data monitor chain by calling DosMonRead.
4. When the spooler receives this special monitor record, it either calls DosPFSAcivate to get information from the Define Code Page (DCP) system files, or it calls the appropriate Presentation Manager routines to get the code page information pertaining to the particular printer. This information is placed into a temporary spooler file.
5. The spooler sends a return code back to the Parallel Port device driver by calling DosMonWrite to place a special monitor record (that is, a Font Monitor Buffer response) into its monitor output buffer registered with the code page chain.

The three printer specific IOCTL commands to support code page and font switching are:

Activate Font: When an application issues a DosOpen for a printer (that is, PRN, LPT1, LPT2, or LPT3), the file system issues an Activate Font IOCTL to the physical Parallel Port device driver to set the active code page and font, according to the active code page of the process making the Open request. At any time, an application can also issue the IOCTL, "Function 48H – Activate Font" on page 18-109 to the physical Parallel Port device driver by using the handle returned from the DosOpen call to open the device. This allows the application to change the active code page and font in the middle of its print job.

When the physical Parallel Port device driver receives the Activate Font IOCTL, if the spooler or another monitor is registered, the Parallel Port device driver sends a Font Monitor Buffer command (in this case, an Activate Font command) to the monitor on the data chain. The monitor receives this special monitor record from its monitor input buffer registered on the data chain by calling DosMonRead. The monitor sends a Font Monitor Buffer response to this command to the physical Parallel Port device driver by calling DosMonWrite, to place a special monitor record into its monitor output buffer registered on the code page chain. If the spooler, or another monitor, is not registered, the physical Parallel Port device driver returns the appropriate return code to the caller of the Activate Font IOCTL.

Query Active Font: When the physical Parallel Port device driver receives the Query Active Font IOCTL request, and if the spooler or another monitor is registered, the physical Parallel Port device driver sends a Font Monitor Buffer command (in this case, a "Function 69H – Query Active Font") to the monitor on the data chain. The monitor receives this special monitor record, from its monitor input buffer registered on the data chain, by calling DosMonRead. The monitor sends a Font Monitor Buffer response to this command to the physical Parallel Port device driver, by calling DosMonWrite, to place a special monitor record into its monitor output buffer registered on the code page chain. The physical Parallel Port device driver then returns the active code page and font information to the caller of the Query Active Font IOCTL. If the spooler, or another monitor, is not registered, the physical Parallel Port device driver returns the appropriate return code to the caller of the Query Active Font IOCTL.

Verify Font: When the physical Parallel Port device driver receives the Verify Font IOCTL request, and if the spooler or another monitor is registered, the physical Parallel Port device driver sends a Font Monitor Buffer command (in this case, "Function 6AH – Verify Font" on page 18-116) to the monitor on the data chain. The monitor receives this special monitor record, from its monitor input buffer registered on the data chain, by calling DosMonRead. The monitor sends a Font Monitor Buffer response to this command to the physical Parallel Port device driver, by calling DosMonWrite, to place a special monitor record into its monitor output buffer registered on the code page chain. The physical Parallel Port device driver then returns the return code to the caller of the Verify Font IOCTL. If the spooler, or another monitor, is not

registered, the physical Parallel Port device driver returns the appropriate return code to the caller of the Verify Font IOCTL.

Refer to “Generic IOCTL Interface” on page 13-8 and “Monitor Dispatcher Notification Interface” on page 13-9 for specific interface parameters.

Character Monitor Performance

The physical Parallel Port device driver character monitor buffer size is 134 bytes. The PRINTMONBUFSIZE = command, within CONFIG.SYS, provides a method for character monitors to increase this size, and thereby increase performance of data to devices connected to the parallel port. The physical Parallel Port device driver allocates and registers its monitor chain buffer based upon the value specified.

The format of this command is:

```
PRINTMONBUFSIZE=xxxx,xxxx,xxxx
```

Where:

xxxx corresponds to LPT1, LPT2, and LPT3 character monitor buffer size, respectively.

The minimum value allowed, for compatibility with existing character monitors, is 134 bytes. The maximum value is 2048 bytes. If a value is specified, which is out of the valid range, a default value of 134 bytes is used.

Character monitors can dynamically determine the size of a device driver's monitor buffer by issuing a DosMonReg call with a 2-WORD buffer:

- The first WORD consists of the length of the buffer (4).
- The second WORD (on return from the call) contains the size of the device driver's monitor chain buffer.

DosMonReg returns with the return code, ERROR_MON_BUFFER_TOO_SMALL. The character monitor then allocates the monitor buffer to the correct size, and reissues the call to DosMonReg.

Parallel Port IRQ Timeout Processing

The physical Parallel Port device driver sets a 120-second timer after receiving a Write request, and sending the first byte to the parallel port. If a hardware interrupt is not generated after 120 seconds, the timer expires, and a timeout occurs.

There are two different mechanisms for processing timeouts:

- If infinite retry is enabled, the physical Parallel Port device driver does not terminate the request, and indefinitely continues to try to send the data to the printer.
- If infinite retry is disabled, the physical Parallel Port device driver terminates the Write request, returning the number of actual bytes sent to the device.

When a monitor is registered, infinite retry is always enabled. The Parallel Port device driver sends a status monitor packet through the data chain (Index = 1), when a timeout error occurs. This activity alerts all monitors of the error.

If a hardware interrupt does occur, the physical Parallel Port device driver resets the timer to 120 seconds, and sends the next byte to the device. This activity continues until all data has been sent to the device. Once the physical Parallel Port device driver determines all characters have been sent, it turns the timer off.

Parallel Port Device Driver Interfaces

The physical Parallel Port device driver has several interfaces:

- Request Packet interface (referred to as the Strategy interface)
- INT 21H interface
- Generic IOCTL interface
- Hardware Interrupt interface
- Monitor Dispatcher Notification interface.

Request Packet Interface

The file system is the primary component which interfaces with the Parallel Port device driver. In response to a function call, the file system creates a request packet containing the information required by the physical Parallel Port device driver to process the request. The file system then calls the physical Parallel Port device driver's strategy entry point, with the registers, ES:BX, containing the address of the request packet.

The Strategy interface uses the CALL/FAR return model, and must preserve the caller's registers. The strategy entry point routes all IOCTL requests and all Open, Read, Write, Close, and Status requests to the appropriate internal Parallel Port device driver routines to handle the requests. For a description and format of a request packet, see Chapter 15, "Physical Device Driver Strategy Commands" on page 15-1.

Request Packet Command Summary: The command code field of a request packet contains the function requested of the physical device driver. The physical Parallel Port device driver is a character device driver, and supports all character device driver functions.

A detailed description of these commands can be found in Chapter 15, "Physical Device Driver Strategy Commands" on page 15-1.

Request Packet Status Description: On the call to the physical Parallel Port device driver, the status code is set to 0. On return from the physical device driver, the status code contains the results of the request (for example, done or error). The physical Parallel Port device driver sets the error bit, busy bit, done bit, and error code, when necessary.

Return Codes: The error codes the physical Parallel Port device driver returns are listed below:

8102H	Device not ready
8103H	Unknown command
8109H	Printer out of paper
810AH	Write fault
810CH	General failure
8111H	Character I/O call interrupted.

The status returned in the request packet for requests the physical Parallel Port device driver does not support is 8103H (Unknown command).

INT 21H Interface

An application running in a DOS Session can access the physical Parallel Port device driver through INT 21H. The file system creates a request packet from the parameters, and calls the physical Parallel Port device driver at its strategy entry point. This interface maintains compatibility with DOS applications. As in the Request Packet interface, the physical Parallel Port device driver preserves the caller's registers, ES:BX, and points to the request packet. The strategy entry point follows the CALL/FAR return model.

Generic IOCTL Interface

The physical Parallel Port device driver supports a set of generic IOCTL functions. Refer to the "Category 5 Parallel Port Control IOCTL Commands" on page 18-105 for a list, and detailed description of these functions.

Note: From the command line, a user can invoke the MODE command to perform the parallel port control functions, *Set/Query Frame Control*, and *Set/Query Infinite Retry*. See the description of the MODE command in the *OS/2 2.0 Command Reference*.

Parallel Port Device Driver Support: All the Category 5 IOCTL commands are supported by the physical Parallel Port device driver. Additionally, the following IOCTLs are also supported by the physical Parallel Port device driver:

Category 0AH Character Monitor Control. Supported by the physical Keyboard, Mouse, and Parallel Port device drivers.

Function 40H Register (A character monitor)

Category 0BH General Device Control. Supported by the all character device drivers.

Function 01H Flush input buffer

Function 02H Flush output buffer

Function 60H Query monitor support.

Character Monitor Control: The IOCTL Category 10 "Function 40H — Register Monitor" on page 18-168 is supported by the physical Keyboard and Mouse device drivers, as well as by the physical Parallel Port device driver. The index field of the data packet received on this call is device specific. That is, it is defined differently for each physical device driver. The index field generally indicates on which monitor chain an application wishes to register a monitor.

As previously discussed, the physical Parallel Port device driver creates two monitor chains for each parallel port device, a data chain and a code page chain. An application issuing this IOCTL function call to a specified parallel port device sets the index parameter as follows:

- *Index=1* Registers a monitor with the data chain.
- *Index=2* Registers a monitor with the code page chain.

Hardware Interrupt (8259) Interface

When a hardware interrupt is pending on an interrupt level owned by the physical Parallel Port device driver, the hardware interrupt manager calls the physical Parallel Port device driver's hardware interrupt entry point for that interrupt level. Family One machines run parallel port devices using hardware interrupt levels IRQ 7 and IRQ 5. The Parallel Port device driver for Family One machines does not share the IRQ levels. PS/2 (Family Two) machines run parallel port devices only on hardware interrupt level IRQ 7. The physical Parallel Port device driver for Family Two machines shares the hardware interrupt level IRQ 7 on PS/2 machines with Micro Channel architecture.

Each of the physical Parallel Port device driver's hardware entry points call a Parallel Port device driver general interrupt routine. This routine sends the next character to the port, issues an EOI (End-Of-Interrupt), and completes a strategy request when the last character has been received by the device.

The hardware interrupt manager saves all registers before calling the Parallel Port device driver interrupt entry points. On return to the hardware interrupt manager, the physical Parallel Port device driver's general interrupt routine clears the carry flag, signalling the interrupt manager that the physical Parallel Port device driver owns the interrupt. If the physical Parallel Port device driver does not own the hardware

interrupt, it sets the carry flag. The physical Parallel Port device driver's hardware interrupt routine follows the CALL/RETURN FAR model, and does not handle nested interrupts.

Monitor Dispatcher Notification Interface

The OS/2 Parallel Port device driver supports character device monitors. For a description of how that support is provided, see "Parallel Port (Printer) Monitors" on page 13-3.

Data sent to each parallel port device can be monitored before reaching the device, by parallel port monitor applications registered with one of two monitor chains. When a parallel port monitor is registered, the physical Parallel Port device driver places all requests (in the form of monitor records or packets) into its monitor chains, by calling the MonWrite device helper routine. Each parallel port monitor registered on the monitor chain can monitor the data destined for the parallel port. Parallel port data that has passed through all monitors registered on the monitor chain is placed into a Parallel Port device driver monitor chain buffer by the OS/2 monitor dispatcher.

Each monitor chain for a parallel port device is created by the Parallel Port device driver, by calling MonitorCreate. For each monitor chain, the physical Parallel Port device driver must provide:

- A device driver monitor chain buffer, into which the monitor dispatcher places the filtered parallel port data, after it has passed through all the parallel port monitors in the monitor chain.

The length of the parallel port device driver's monitor chain buffer is 134 bytes. This length is specified by the physical Parallel Port device driver in the first WORD of the buffer, when MonitorCreate is called. The length of parallel port monitor input and output buffers, registered with a parallel port monitor chain, must be greater than, or equal to, 134 bytes plus 20 bytes.

- The address of a notification routine that is called by the monitor dispatcher when it has placed filtered data into the monitor chain buffer.

Before the monitor dispatcher calls the physical Parallel Port device driver's notification routine, it:

1. Places a data record in the physical Parallel Port device driver's monitor chain buffer starting at the second WORD of the buffer
2. Places the length of the data record (in bytes) in the first WORD of the physical Parallel Port device driver's monitor chain buffer
3. Sets the register pair, ES:SI, to point to the Parallel Port device driver's monitor chain buffer.

The general monitor packet format for code page processing is described below:

Field	Length
Monitor Flags	WORD
System File Number	WORD
Command Byte	BYTE
Reserved. Set to 0.	BYTE
Reserved. Set to 0.	BYTE
Reserved. Set to 0.	BYTE
Return Code	WORD
Code Page	WORD
Font ID	WORD

Monitor Packet for Code Page Processing

parallel port PDD

The general monitor packet format for Open, Write, Close, Status, and Print-Job-Title monitor requests is described below:

Field	Length
Monitor Flags	WORD
System File Number	WORD
Data to be Monitored (optional)	128 BYTES

General Monitor Packet

Data items included in these data structures are defined in the descriptions of the corresponding request packet data structures. The physical Parallel Port device driver communicates with parallel port monitors using the monitor protocol.

Parallel Port Monitor Buffer Command

Each monitor record can be of variable length, and contain a WORD of flags defining the type of monitor packet. These flags are defined in Figure 13-1.

Note: Monitor packets that parallel port monitors do not understand should be returned to the physical Parallel Port device driver on the monitor chains from which they were received.

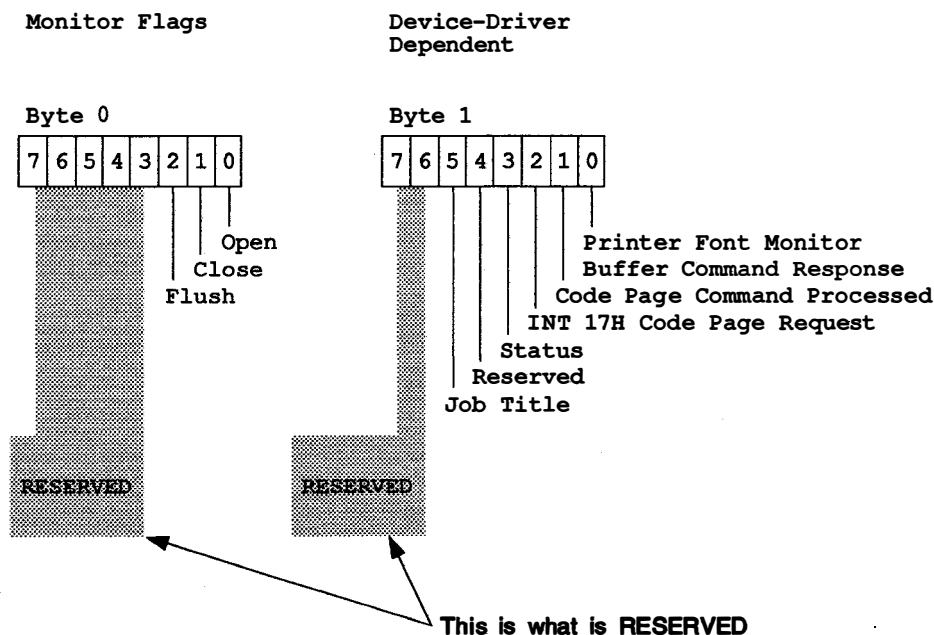


Figure 13-1. Monitor Packet Flags

Monitor Open Packet: When the Open bit is set to 1 (byte 0, bit 0), the monitor buffer is an open packet from the physical Parallel Port device driver. In this case, the next two bytes are:

Field	Length
System File Number	WORD

Monitor Close Packet: When the Close bit is set to 1 (byte 0, bit 1), the monitor buffer is a close packet from the physical Parallel Port device driver. In this case, the next two bytes are:

Field	Length
System File Number	WORD

Monitor Write Packet: When the Close bit is set to 0 (byte 0, bit 1), and the Open bit is set to 0 (byte 0, bit 0), the monitor buffer is a write data packet from the physical Parallel Port device driver. In this case, the next bytes are:

Field	Length
System File Number	WORD
Data to be Monitored	128 BYTES

Font Monitor Packet: When the Font Monitor Buffer Command/Response bit is set to 1 (byte 1, bit 0), the monitor buffer is a Font Monitor Buffer command. In this case, the next six bytes are:

Field	Length
System File Number	WORD
Command Byte	BYTE
Reserved. Set to 0.	BYTE
Reserved. Set to 0.	BYTE
Reserved. Set to 0.	BYTE

Byte 2-3 System File Number

Byte 4 Font Monitor Buffer Command byte, which indicates the type of command or response.

Byte 5 Reserved. Must be set to 0.

Byte 6-7 Reserved. Must be set to 0.

When the Code Page Command Processed bit is set to 1 (byte 1, bit 1), the Font Monitor Buffer command has been processed by the spooler.

The physical Parallel Port device driver sends Font Monitor Buffer commands to its monitors through the data monitor chain (Index = 1). A parallel port monitor, which services the Font Monitor Buffer command sets the Code Page command processed bit, and uses DosMonWrite to place the Font Monitor Buffer response into the code page monitor chain (Index = 2). Data to be printed continues to flow on the data monitor chain. The code page monitor chain is used only for the Font Monitor Buffer responses, so that the physical Parallel Port device driver does not block a program issuing a code page and font IOCTL request, when print data monitor buffers are already queued ahead of the IOCTL request.

Font Monitor Buffer Commands: The valid Font Monitor Buffer commands, along with the remainder of the buffer contents for the responses, are as follows:

Byte 4 = 01H Activate Font. Command data, starting at byte 8:

Field	Length
Return Code	WORD
Code Page	WORD
Font ID	WORD

parallel port PDD

Return Code A value returned, starting at byte 8, and includes the following values:

0000H	Successful completion
0002H	Code page is not available
0003H	No code page function, because spooler not started
0004H	Font ID is not available (verify)
0009H	Error caused by switcher error, not by input parameters
000AH	Error caused by invalid printer name as input
000DH	Received code page request, when code page switcher not initialized
000FH	SFN table full cannot activate another entry
0013H	DASD error reading font file
0015H	DASD error reading font file definition block
0017H	DASD error, while writing to temporary spool file
0018H	Disk full error, while writing to temporary spool file
0019H	Spool file handle was bad.

Code Page The value of the code page to make the currently active code page.

0000H	If the Code Page value and Font ID are specified as 0, set printer to hardware default code page and font.
0001H-FFFFH	Valid code page numbers.

Font ID The ID value of the font to make currently active.

0000H	If the Code Page value and Font ID are specified as 0, set printer to hardware default code page and font. If only the Font ID is 0 and the Code Page is a valid non-zero, then any font within the specified code page is acceptable.
0001H-FFFFH	Valid Font ID numbers, font types defined by the font file definitions for fonts that can be downloaded. For cartridge fonts, Font IDs are the numbers on the cartridge label and are also entered in the DEVINFO statement for the printer.

The physical Parallel Port device driver passes the Font Monitor Buffer command to the monitor on the data monitor chain (Index = 1). The monitor returns the Font Monitor Buffer response to the Parallel Port device driver on the code page monitor chain (Index = 2).

Byte 4 = 02H Query Active Font. There is no additional command data. Data returned from this function call includes:

Field	Length
Return Code	WORD
Code Page	WORD
Font ID	WORD

Return Code A value returned, starting at byte 8, which includes the following values:

0000H	Successful completion
0003H	No code page function, because spooler not started
0009H	Error caused by switcher error, not by input parameters
000AH	Error caused by invalid printer name as input
000DH	Received code page request, when code page switcher not initialized
0010H	Received request for SFN not in SFN table.

Code Page On return, is set to currently active code page:

0000H If the Code Page value and Font ID are returned as zero, the printer is set to the hardware default code page and font.

0001H-FFFFH Valid code page numbers.

Font ID On return, is the ID value of the Font, which is currently active:

0000H If the Code Page value and Font ID are specified as zero, the printer is set to the hardware default code page and font. A Font ID value of zero can indicate the default font of a particular code page.

0001H-FFFFH Valid Font ID numbers, font types defined by the font file definitions for fonts that can be downloaded. For cartridge fonts, Font IDs are the numbers on the cartridge label, and are also entered in the DEVINFO statement for the printer.

The physical Parallel Port device driver passes the Font Monitor Buffer command to the monitor on the data monitor chain (Index=1). The monitor returns the Font Monitor Buffer response to the Parallel Port device driver on the code page monitor chain (Index=2).

Byte 4 = 03H Verify Font. Command Data, starting at byte 8:

Field	Length
Return Code	WORD
Code Page	WORD
Font ID	WORD

Return Code A value returned, starting at byte 8, and includes the following values:

0000H Successful completion
0002H Code page is not available
0003H No code page function, because spooler not started
0004H Font ID is not available (verify)
000AH Error caused by invalid printer name as input
000DH Received code page request, when code page switcher not initialized.

Code Page The Code Page number to validate.

0000H-FFFFH Valid code page numbers.

Font ID The Font ID value to validate.

0000H-FFFFH Valid Font ID numbers.

The physical Parallel Port device driver passes the Font Monitor Buffer command to the monitor on the data monitor chain (Index=1). The monitor returns the Font Monitor Buffer response to the Parallel Port device driver on the code page monitor chain (Index=2).

Monitor Status Packet: When the Status bit is set to 1 (byte 1, bit 3), this indicates that the monitor buffer is a status packet from the Parallel Port device driver. In this case, the next four bytes are:

Field	Length
System File Number	WORD
Error Code	BYTE
Reserved	BYTE

parallel port PDD

The physical Parallel Port device driver sends either *device not ready* (02H), *printer out of paper* (09H), or *write fault* (0AH) as the error code.

Monitor-Job Title Packet: A print-monitor job title packet passes the name of the file being printed to all monitors registered with the physical Parallel Port device driver.

When the job title bit is set to 1 (byte 1, bit 5), it indicates the monitor buffer is a job title packet from the Parallel Port device driver. In this case, the next bytes are:

Field	Length
System File Number	WORD
Job Title Length	WORD
Job Title Buffer	BYTES

The Job Title Length field includes the null character. The Job Title Buffer is terminated with the null character. The Job Title Buffer field contains bytes 6 – 131.

Chapter 14. Physical Video Device Drivers

Video devices are accessed by using Base Video Handlers (BVHs). These BVHs consist of one or more Dynamic Link Libraries (DLLs). In the representative case of the VGA, BVHVGA.DLL manages the device for full-screen sessions, while DISPLAY.DLL (renamed from IBMVGA.DLL) manages the device for the Presentation Manager interface. Although these device handlers are initialized by different sections of the system at this time, they are architecturally compatible and can easily be combined at a later date.

Video Device Handler Identification

The list of the active video device handlers and their components resides in CONFIG.SYS as environment variables. To conserve environment space, these variables will be removed from the environment during Shell Initialization. The value of the VIDEO_DEVICES environment variable is the list of the names of the environment variables that describes each of the video device handlers. Commas are used to separate the names in this list. The following is an example of how to specify the environment variables, ARTICHOKE and WATERMELON, as those defining the active video device handlers, with ARTICHOKE used as configuration number one, and WATERMELON used as configuration number two.

```
SET VIDEO_DEVICES=ARTICHOKE,WATERMELON
```

The value of each environment variable, which describes a video device handler, is composed of three keywords and the values associated with them. These keywords are separated by blanks and can be specified in any order, and in any combination of upper and lower case characters. The DEVICE() keyword defines the list of names of the dynamic link libraries and physical device drivers, which are combined to create the video device handler. The names are separated by commas, and their order determines the order in which the components will be initialized. These names represent only those parts of the BVH, which need to be called to initialize the Call Vector Table. That is, physical device drivers should not be included in the list if they are only called by the dynamic link libraries, and do not directly modify the Call Vector Table.

The default initialization entry point name for the dynamic link libraries is DEVENABLE. An alternate entry point name can be specified by following any DLL name with +AltName, where *AltName* is the entry point name.

The default initialization IOCTL for physical device drivers is "Function 73H – Initialize Call Vector Table" on page 18-58. An alternate function number and category can be specified by following the device name by +Func+Cat, where *Func* is the the function number, and *Cat* is the category. Both numbers must be specified in hexadecimal.

The following is an example of a device handler that is composed of two physical device drivers, DEV1 and DEV2, and three dynamic link libraries, DYN1 through DYN3. The second physical device driver uses an alternate initialization IOCTL, and the third dynamic link library uses an alternate initialization entry point. POINTER\$ is the default physical Pointer device driver.

```
SET ARTICHOKE=DEVICE(DEV1,DYN1,DEV2+74+05,DYN2,DYN3+0THERENT)
```

DosOpen is called for each name in the list to check if it is a device driver with an associated DEVICE= statement in CONFIG.SYS. If the call fails, DosLoadModule is called to check if it is a dynamic link library. If both of these calls fail for any name in the list, the entire device is ignored.

If the optional PTRDEVP() keyword is specified, it defines the names of the physical Pointer device drivers. If it is not specified, it defaults to PTRDEVP(POINTER\$). The following is an example of a device with only one dynamic link library component and a unique physical Pointer device driver for protect mode.


```
SET WATERMELON=DEVICE(SEEDLESS) PTRDEVP(PPOINT)
```

Note: This design is not limited strictly to physical video devices. By writing a device handler, video data could be written to any device, such as a printer or a plotter. In addition, by using alternate initialization entry points, multiple devices can be handled by the same physical device handler.

All of the video device handlers shipped with OS/2 2.0 are dynamic link libraries. They can be defined by the following environment variables, which use the default keywords of PTRDEVP(POINTER\$) and PTRDEVR(POINTER\$).

```
SET VIO_IBMMPA=DEVICE(BVHMPA)
SET VIO_IBMCGA=DEVICE(BVHCGA)
SET VIO_IBMEGA=DEVICE(BVHEGA)
SET VIO_IBMVGA=DEVICE(BVHVGA)
SET VIO_IBM8514A=DEVICE(BVHVGA,BVH8514A)
SET VIO_IBM8514A=DEVICE(BVH8514A)
SET VIO_IBMXGA=DEVICE(BVHXGA)
```

The following statements define a system with an 8514 display attached to an 8514/A, as the only active video device.

```
SET VIDEO_DEVICES=VIO_IBM8514A
SET VIO_IBM8514A=DEVICE(BVHVGA,BVH8514A)
```

However, the statement below defines a system with an 8514 display attached to an 8514/A, and another PS/2 display attached to the VGA connector, as independent video devices.

```
SET VIDEO_DEVICES=VIO_IBMVGA,VIO_IBM8514A
SET VIO_IBMVGA=DEVICE(BVHVGA)
SET VIO_IBM8514A=DEVICE(BVH8514A)
```

Two other device handlers are provided with OS/2 2.0. The names of these device drivers are fixed. The Base Video Subsystem loads them automatically as they are needed.

BVHINIT.DLL is the generic device handler used by System Installation and System Initialization. It provides the minimum function necessary to support installation of the system and reporting of system errors, during startup. It is loaded only if no other BVHs are successfully loaded.

BVHWNDW.DLL is the device handler that can support VIO window sessions, and provides the interface to the Presentation Manager interface by treating the PM interface as a virtual Video device driver. BVHWNDW.DLL is loaded only for VIO window sessions.

Video Device Chaining

Video device handlers (BVHs) can be chained together when multiple BVHs share the responsibility of supporting a specific video adapter. This is accomplished by allowing previously loaded BVHs to attempt the handling of BVH functions. BVH8514A and BVHXGA are chained BVH's, shipped with the product that provides this support.

A VGA and 8514A Scenario

During system initialization, BVHVGA is first called to initialize the Call Vector Table, that is, the table used by BVS to give control to the BVH routines. BVHVGA initializes as though no other BVH will be handling the device. Next, BVH8514A is called to initialize the same Call Vector Table. However, BVH8514A saves a copy of the Call Vector Table before changing it.

When calls are made to this chained BVH, BVH8514A receives the call, and passes it to the BVHVGA routine through the saved Call Vector Table. If an error occurs, or if the results of the routine need to be modified, BVH8514A handles the call. Thus, BVH8514A uses the BVHVGA routines to perform all common

functions. *Device chaining* can be viewed as a mechanism to allow one BVH to filter function calls to another BVH.

Primary Display Identification

The primary display is the default display chosen by VIO for full-screen sessions. It is also the display on which VIO and hard error pop-ups are shown. Notice that the primary display used by VIO is not necessarily the display on which the Presentation Manager environment runs. The Presentation Manager interface normally runs on the highest resolution display. In a dual display configuration, the highest resolution display is not necessarily a display, which can be used in text mode.

The primary display is the pop-up display. A physical Video device driver must determine if it represents the pop-up display. If so, the Video device driver must specify that it represents the pop-up display configuration in Query Config Info.

WrtToScrn/Panic Write Support

Before the video subsystem is loaded, and when the system is about to abnormally terminate, messages are sent to the screen by the WrtToScrn function, also known as Panic Write. This function switches to real mode, and executes the INT 10H function to set the video mode to a text mode (BIOS mode 3). It then uses the BIOS INT 10H, "Write TTY" to display the message.

However, adapters, such as the 8514/A, have Native modes which cannot be changed by INT 10H. In such cases, the BVH must include a Level 0 device driver, which hooks INT 10H to provide extended set mode support. This hook must force the adapter out of its native mode, then pass control to the previous INT 10H support. If these conditions are not satisfied, the adapter should not drive the power up display.

Running System Installation

The generic device handler, BVHINIT.DLL, is primarily used by System Installation. It is also used for those situations when no video devices have been identified in CONFIG.SYS. It provides only the functions required for System Installation, and is otherwise device independent.

Through VioGetConfig, it reports the highest function video adapter and display that it can identify. That is, it can identify only the MPA, CGA, EGA, VGA, 8514A, and XGA, along with their respective displays. Although it does not support mode and font setting, it attempts to load the 850 code page for the current EGA or VGA mode during System Initialization.

If no video devices have been identified in CONFIG.SYS, the generic device handler is loaded, by trying to load each of the following devices until one is successfully loaded:

```
GENERIC=DEVICE(BVHVGA,BVH8514A)
GENERIC=DEVICE(BVHVGA)
GENERIC=DEVICE(BVHEGA)
GENERIC=DEVICE(BVHCGA)
GENERIC=DEVICE(BVHINIT)
```

Using Loadable Device Drivers

Loadable device drivers present a unique problem when used to support video devices, because some video support can be required before the DEVICE = statements have been processed by System Initialization. To handle this problem, two different initialization calls are made.

The Enable Subfunction determines which type of initialization is to be done. If the subfunction is 1 (Fill Logical Device Block), the DevEnable function is requested to add all of the functions supported by the device handler to the Call Vector Table. If the subfunction is 3 (Fill Initialization Device Block), the

InitEnable function is requested to add only those functions, which can be supported without the use of a loadable device driver to the Call Vector Table.

Regardless of the success of the InitEnable function, the DevEnable function is called at Shell Initialization time. See "DevEnable" on page 14-5, and "InitEnable" on page 14-7 for more information.

Video Device Handler Interfaces

The functions that follow are video primitives reserved for use by OS/2 system components. Unexpected results can occur, if these functions are invoked by applications. All of the video device handler functions described below (except for the DLL Initialization function) use the same calling sequence. Parameters are passed to the routines on the stack. The entry point is found in the BVH Call Vector Table at the index of the function number. The calling sequence used to invoke all the routines is as follows:

PUSH@	OTHER	Environment	; Environment buffer
PUSH@	OTHER	ParmBlock	; Parameter block
PUSH	DWORD	Function	; Function number
CALL	FAR	BVH Routine Entry Point	

Environment: The environment buffer. The format of this buffer is defined by the BVH developer. The selector is a huge selector.

ParmBlock: A data structure containing all of the parameters of the operation to be performed. The general format of this structure is:

```
Length of parameter block in bytes (=NN)  WORD
Flags                                     WORD
```

Bit 0 indicates whether the physical hardware is updated.

Off = Initialize only the environment buffer

On = Initialize the environment buffer and the hardware state.

Bits 1 thru 15 are reserved and must be Off.

The *length of the parameter block* is the total length, including the length field itself. The number in parentheses (=NN) represents the value of this field, if it is a constant.

The first flag bit always indicates a background verses foreground state for this function. If the bit is *on*, the adapter is actually updated, as it is when an application is in the foreground. If the bit is *off*, only the buffer that is used to shadow the adapter when an application is in the background is actually updated. All bits that are not currently defined as reserved must be *off*.

Function: The function identifier for this routine. This corresponds to offset into the Call Vector Table, and can be used to determine the number of parameters on the stack. This is consistent with the existing DDI used by Presentation Manager interface. All routines are expected to return with AX=0, if no error was detected. Otherwise, an error code is returned in AX. The following errors are common to all commands:

ERROR_VIO_INVALID_LENGTH, if the parameter block length is incorrect.

ERROR_VIO_INVALID_PARAMS, if a reserved flag bit is non-zero, or if the function number does not match the routine.

DevEnable

This function fills the entries in the Call Vector Table for all of the functions supported by this BVH. It is called as a subfunction of the Presentation Manager enable function entry point. To initialize the Call Vector Table for dynamic link libraries:

```

PUSH  DWORD  Parameter2    ; Variable parameter 2
PUSH  DWORD  Parameter1    ; Variable parameter 1
PUSH  DWORD  Subfunction    ; Enable subfunction

CALL  FAR     DevEnable

```

Parameters

Parameter2 for Subfunction = 1: A far pointer to this structure:

```

DWORD  Flags Pointer
DWORD  Call Vector Table

```

Flags Pointer: Pointer to where the flags controlling calls to the Fill Physical Device Block function go.

Call Vector Table: Pointer to the default dispatch table containing the addresses of the default handler functions, and the functions supported by this component. Each entry in the Call Vector Table is the far address of a Video Device Handler function, which must be callable from both Ring 2 and Ring 3. The far address of the *n*th BVH function is the *n*th DWORD in the table, beginning with Function 0.

The functions listed below are defined as follows:

000H-0FFH (000-255) Reserved for Presentation Manager interface
100H-11FH (256-287) Reserved for BVS
120H-12FH (288-303) Reserved for IBM JAPAN
130H-13FH (304-319) Reserved for MSKK.

100H (256) Text Buffer Update (see page 14-8)
101H (257) Initialize Environment (see page 14-11)
102H (258) Save Environment (see page 14-12)
103H (259) Restore Environment (see page 14-14)
104H (260) Query Config Info (see page 14-15)
105H (261) Query DBCS Display Info (see page 14-16)
106H (262) Query Color Lookup Table (see page 14-17)
107H (263) Set Color Lookup Table (see page 14-18)
108H (264) Query Cursor Info (see page 14-19)
109H (265) Set Cursor Info (see page 14-20)
10AH (266) Query Font (see page 14-21)
10BH (267) Set Font (see page 14-22)
10CH (268) Query Mode (see page 14-23)
10DH (269) Set Mode (see page 14-24)
10EH (270) Query Palette Registers (see page 14-25)
10FH (271) Set Palette Registers (see page 14-26)
110H (272) Query Physical Buffer (see page 14-27)
111H (273) Free Physical Buffer (see page 14-28)
112H (274) Query Variable Info (see page 14-29)
113H (275) Set Variable Info (see page 14-31)
114H (276) Terminate Environment (see page 14-33)
115H (277) Print Screen (see page 14-34)
116H (278) Write TTY (see page 14-35)
117H (279) Query LVB Info (see page 14-36)

Parameter1 for Subfunction = 1: Far pointer to this structure:

DWORD Engine Version
DWORD Count of Table Functions

Engine Version: Version of the Presentation Manager Graphics Engine.

Count of Table Functions: The number of entries in the passed dispatch table. The driver can only write this many entries into the table.

Subfunction: The Presentation Manager enable subfunction number. Its value must be 1 to invoke the Presentation Manager Fill Logical Device Block subfunction. All pieces of the BVH must support this function. Pieces of the BVH that do not support the Presentation Manager interface do not need to support the other functions. Any function not supported will return PMERR_DEV_FUNC_NOT_INSTALLED.

Remarks: This function is supported by the video dynamic link functions, and is called only once for each adapter supported by the physical device driver. A video device handler determines if the display adapter, and that which the adapter supports, is present. If not present, this function returns an error. Every part of a BVH must successfully initialize the Call Vector Table for that device to be usable by the OS/2 operating system.

InitEnable

This function fills the entries in the Call Vector Table for all of the functions supported by this BVH using only SCREEN\$ device driver. It is called with parameters similar to the DevEnable entry point specified above, except for the Subfunction parameter.

To initialize the Call Vector Table for dynamic link libraries:

```
PUSH  DWORD   Parameter2   ; Variable parameter 2
PUSH  DWORD   Parameter1   ; Variable parameter 1
PUSH  DWORD   Subfunction   ; Enable subfunction

CALL  FAR     InitEnable
```

Parameters

Parameter2 for Subfunction = 3: See “DevEnable” on page 14-5.

Parameter1 for Subfunction = 3: See “DevEnable” on page 14-5. Subfunction is the Presentation Manager enable subfunction number. Its value must be 3 to invoke the Presentation Manager Fill Initialization Device Block subfunction. The default entry point of DevEnable can be overridden by specifying an alternate name in the DEVICE() parameter describing this BVH in CONFIG.SYS.

Remarks: This function is called only if video functions are required before Shell Initialization.

The BVH must be able to support the following subfunctions:

100H (256)	Text Buffer Update
101H (257)	Initialize Environment
102H (258)	Save Environment (settable environment only)
103H (259)	Restore Environment (settable environment only)
104H (260)	Query Config Info
105H (261)	Query DBCS Display Info
108H (264)	Query Cursor Info
109H (265)	Set Cursor Info
10CH (268)	Query Mode (80x25 text or equivalent only)
10DH (269)	Set Mode (80x25 text or equivalent only)
112H (274)	Query Variable Info (code page only)
113H (275)	Set Variable Info (code page only)

See “DevEnable” on page 14-5.

Text Buffer Update

This function performs text updates to the logical and physical video buffers. All references to the buffer are made through the text row and column of each cell affected by the called function.

Subfunction Number: 100H (256)

Parameter Block Format

Size	Description	Size	Description
----	-----	----	-----
WORD	ParmLength	WORD	TouchXLeft
WORD	Flags	WORD	TouchYTop
DWORD	AppDataAddr	WORD	TouchXRight
DWORD	SecondAddr	WORD	TouchYBottom
WORD	Index	WORD	LVBRowOff
WORD	StartRow	WORD	LVBColOff
WORD	StartCol	WORD	LVBWidth
WORD	SecondRow	WORD	LVBHeight
WORD	SecondCol	BYTE	LVBFormatID
WORD	RepeatFactor	BYTE	LVBAtrCount
WORD	LogicalBufSel		

ParmLength: Length of the data structure in bytes (greater than, or equal to, 26), including the Length field itself. The maximum length is 44 bytes. Values not passed are assumed to be the default values for the environment.

Flags: Defined as follows:

Bit 0 indicates whether the physical video buffer needs to be updated.

Off = The physical video buffer must not be updated.

On = The physical video buffer must be updated.

Bit 1 indicates whether the logical video buffer needs to be updated.

Off = The logical video buffer may optionally be updated.

On = The logical video buffer must be updated.

Bit 2 indicates that attribute information in the user buffer is in CGA format and may need to be translated into the format used by the device. This bit is set for VioWrtTTY calls any time that ANSI is active, since ANSI only recognizes CGA format attributes.

Off = Use attributes in existing format.

On = Translation to or from CGA format required.

Bits 3-15 are reserved and must be Off.

Selector/Offset of the Application Data Area: Pointer to the application's data area, which provides either the source or the destination for the buffer operation.

Selector/Offset of Secondary Data Area: Pointer to the additional parameter required by this type of update operation. It is used to define, at most, one cell of information which is used repetitively as filler. This is used only with Index = 3, 4, 5, 6, or 9.

Index: Defined as follows:

0 = Read Cell Types from (Row,Col) as word of flags:

Bit 0 indicates whether the cell is part of a single or double cell character.

Off = The cell represents a single cell character.

On = The cell represents part of a double cell character. Bit 1 is examined for more information.

Bit 1 indicates whether the cell is a trailing cell of a double cell character.

Off = The cell represents the leading or only cell of a character.

On = The cell represents the trailing cell of a double cell character.

Bits 2 thru 15 are reserved and must be Off.

1 = Read Characters from (Row,Col)

2 = Read Cells from (Row,Col)

3 = Scroll (Row,Col) through (Row2,Col2) Up

4 = Scroll (Row,Col) through (Row2,Col2) Down

5 = Scroll (Row,Col) through (Row2,Col2) Left

6 = Scroll (Row,Col) through (Row2,Col2) Right

7 = Write Cells to (Row,Col)

8 = Write Characters to (Row,Col)

9 = Write Characters with Constant Attr to (Row,Col)

10 = Write Repeated Character to (Row,Col)

11 = Write Repeated Attribute to (Row,Col)

12 = Write Repeated Cell to (Row,Col)

13 = Copy LVB Rect to PVB

Starting Row/Column: Defines the text location in the Video Buffer at which the update is to be started. For Function 13, this is the upper-left corner of the rectangle to be written.

Secondary Row/Column: Defines the text location in the Video Buffer to which cells will be moved. This is used only when moving cells from one location to another (Index=3–6). For Function 13, this is the lower-right corner of the rectangle to be written.

Repeat Factor: Repeat factor used when updating the buffer. It represents the number of character cells to be updated, or the number of rows or columns to be scrolled. This is used for both input and output.

Logical Buffer Selector: Selector used for the beginning of the logical video buffer. This selector is a huge selector with a maximum size of 1MB.

TouchXLeft: The x-coordinate of the upper-left corner of the tightest rectangle, which circumscribes the cells touched by the current function. If no cells were touched (as in a Read), -1 is returned. Collectively, this field (and the next three fields) form an area of influence for the call. This field is returned by the BVH.

TouchYTop: The y-coordinate of the upper-left corner of the tightest rectangle, which circumscribes the cells touched by the current function. If no cells were touched, -1 is returned. This field is returned by the BVH.

TouchXRight: The x-coordinate of the lower-right corner of the tightest rectangle, which circumscribes the cells touched by the current function. If no cells were touched, -1 is returned. This field is returned by the BVH.

TouchYBottom: The y-coordinate of the lower-right corner of the tightest rectangle, which circumscribes the cells touched by the current function. If no cells were touched, -1 is returned. This field is returned by the BVH.

LVBRowOff: The row offset of the upper-left corner of the LVB in PVB coordinates. All reads, writes and scrolls are done in PVB coordinates.

LVBColOff: The column offset of the upper-left corner of the LVB in PVB coordinates. All reads, writes and scrolls are done in PVB coordinates.

LVBWidth: The width of the LVB in cells. Must be > 0.

LVBHeight: The height of the LVB in cells. Must be > 0.

LVBFormatID: The format ID of the LVB. If this value and attribute count are both 0, the Format ID and attribute count for the current mode are used.

LVBAttrCount: The attribute count for the LVB.

Remarks: The Touchxxxx fields circumscribe the area of the LVB, or PVB, for a rectangle that was potentially changed by the given operation. For example, a Write that included the cells (10,12) to (79,12), (0,13) to (79,13), and (0,14) to (8,14) returns TouchXLeft=0, TouchYTop=12, TouchXRight=79, and TouchYBottom=14. Any new functions added to the BVH interface that can affect the data in the PVB or LVB must include a return area for the rectangle of the video buffer that was affected by the given call.

The LVBxxxx fields indicate that an LVB can differ from the normal LVB format. The information about the LVB is taken from these fields, if they are included. Notice that these fields allow an LVB to begin at a location other than (0,0) and allow LVBs of different row and column dimensions.

The Text Buffer Update routine returns with AX=0, if no error was detected. Otherwise, the following error codes are returned in AX:

ERROR_VIO_COL,	if an invalid column number was specified
ERROR_VIO_INVALID_LENGTH,	if the ParmLength was incorrect
ERROR_VIO_INVALID_PARMS,	if the Index was incorrect
ERROR_VIO_MODE,	if updates are not supported in the current video mode
ERROR_VIO_ROW,	if an invalid row number was specified.

Initialize Environment

This function causes the Environment Buffer and the video adapter (optional) to be initialized.

Subfunction Number: 101H (257)

Parameter Block Format

Length of parameter block in bytes (=6) passed on input WORD

Flags WORD

Bit 0 indicates whether the physical hardware is updated.

Off = Initialize only the environment buffer.

On = Initialize the environment buffer and the hardware state.

Bit 1 indicates whether the 3xBox is being initialized.

Off = The 3xBox is not being initialized.

On = The 3xBox is being initialized.

Bits 2 thru 15 are reserved and must be Off.

Logical Video Buffer selector. A huge selector. WORD

Remarks: It must be possible to call Restore Environment before Save Environment. This routine always returns with AX=0.

Save Environment

This function is used to save all aspects of the video adapter, including the hardware state and the video buffers.

Subfunction Number: 102H (258)

Parameter Block Format

Length of parameter block in bytes (=6) passed on input	WORD
Flags	WORD
<p>Bit 0 is reserved and must be Off.</p> <p>Bit 1 indicates whether hardware state (mode, CLUT, everything except the buffer) is saved.</p> <p>Off = The hardware state is not saved. On = The hardware state is saved.</p> <p>Bit 2 indicates whether the physical display is fully saved for session switching.</p> <p>Off = The physical display is not fully saved. On = The physical display is fully saved.</p> <p>Bit 3 indicates whether the physical display is partially saved for pop-ups.</p> <p>Off = The physical display is not partially saved. On = The physical display is partially saved.</p> <p>Bits 4 thru 15 are reserved and must be Off.</p>	
Logical Video Buffer selector. A huge selector.	WORD

Remarks: Bits 2 and 3 are mutually exclusive. If both are specified, bit 3 will be ignored. Bit 4 is used in combination with bits 2 and 3. The code and data segments referenced or accessed to perform the functions selected by bits 1 and 3 must be locked during device driver initialization. The format of the data saved in the segments passed as input is determined by the device handler.

Partial saves are invoked on VIO and *hard error pop-ups*. Pop-ups are displayed on the primary display configuration. The device driver must save whatever portion of the physical display buffer will be overlaid by the pop-up. To display a pop-up, OS/2 2.0 switches to the highest resolution 80x25 text mode supported by the primary display configuration (mode 3 or 7, whichever is listed first in the list of modes supported by the display configuration). Alternatively, if a device driver's physical display buffer is not overlaid by a pop-up, the physical device driver returns zero for partial save size.

When a hard error pop-up occurs before a VIO pop-up has cleared, the Save Environment function is called twice before the Restore Environment is called. Therefore, the device handler must be prepared to handle both a partial save of a graphics mode and a full save of the text mode of the user pop-up.

OS/2 2.0 allocates the buffer, in which the physical display buffer is saved by using DosAllocHuge. The selector to the Data Packet addresses the first of *n* segments in which the physical display buffer is saved. (The offset to the Data Packet should be ignored.) The selector to the second segment can be calculated by adding the DosAllocHuge increment to the first selector value. The third selector can similarly be calculated by adding the DosAllocHuge increment to the second selector value, and so forth. Enough

selectors are allocated to meet the full/partial buffer requirement specified by the physical device driver. The selectors each address 64KB except the last selector, which addresses the remainder.

The Save Environment routine returns with AX=0, if it can successfully save the environment to the Environment Block, and the Logical Video Buffer. Otherwise, it returns with AX = ERROR_VIO_MODE.

Restore Environment

This function is used to restore all aspects of the video adapter, including the hardware state and the video buffers.

Subfunction Number: 103H (259)

Parameter Block Format

Length of parameter block in bytes (=6) passed on input WORD

Flags WORD

Bit 0 is reserved and must be Off.

Bit 1 indicates whether hardware state (mode, CLUT, everything except the buffer) is saved.

Off = The hardware state is not restored.

On = The hardware state is restored.

Bit 2 indicates whether the physical display is fully restored for session switching.

Off = The physical display is not fully restored.

On = The physical display is fully restored.

Bit 3 indicates whether the physical display is partially restored for pop-ups.

Off = The physical display is not partially restored.

On = The physical display is partially restored.

Bits 4-15 are reserved and must be Off.

Logical Video Buffer selector. A huge selector. WORD

Remarks: The Restore Environment routine returns with AX=0, if it can successfully restore the environment from the Environment Block and the Logical Video Buffer. Otherwise, it returns with AX = ERROR_VIO_MODE. See "Save Environment" on page 14-12 for more information.

Query Config Info

This function returns all of the information necessary to identify the current video adapter and display.

Subfunction Number: 104H (260)

Parameter Block Format

Length of parameter block in bytes (=8) passed on input	WORD
Flags. Must be zero	WORD
Far Address of the Config data structure defined by VioGetConfig.	DWORD

Remarks: The Environment Buffer is not used by this function. The Environment Buffer address is passed as a DWORD of zero. If the Length specified in the Config Data is larger than the maximum possible length, or if the Length is specified as 2 (the length of Length field itself), it is replaced by the largest valid length. This function returns with AX=ERROR_VIO_INVALID_LENGTH, only if the Length specified is less than 2.

Query DBCS Display Info

This function returns various forms of DBCS information used by the display.

Subfunction Number: 105H (261)

Parameter Block Format

Length of parameter block in bytes. If Length is specified as 2, only the maximum length of the parameter block is returned in the length field. If the length is not 2, it defines the maximum amount of data returned. WORD

Flags. Must be zero. WORD

Length of double cell character table WORD

Offset of double cell character table, which consists of WORD pairs that define the low and high limits (inclusive) of ranges of double cell characters. WORD

Remarks: This function is used to get the DBCS display information associated with the given environment buffer, and returns with AX=ERROR_VIO_INVALID_LENGTH if the Length specified is less than 2 or the buffer was too short to return all of the DBCS display information. Otherwise, Query DBCS Display Info returns with AX=0.

Query Color Lookup Table

This function reads the definitions of the colors from the Color Lookup Table.

Subfunction Number: 106H (262)

Parameter Block Format

Length of parameter block in bytes (=12) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical hardware is to be read.	
Off = Return data from the environment buffer only.	
On = Read the hardware to update the environment buffer before returning the requested data.	
Bits 1-15 are reserved and must be Off.	
Far address of Color Lookup Table. The Table format is device-dependent. Three-byte table entries are returned for the VGA. Each table entry contains a red, green, and blue color index, respectively.	DWORD
Index of first table entry to get.	WORD
Number of table entries to get. Three-byte table entries are returned for the VGA. Each table entry contains a red, green, and blue color index, respectively.	WORD

Remarks: This function with AX=0, if it can successfully get all of the requested registers from the Color Lookup Table. Otherwise, this routine returns with AX = ERROR_VIO_INVALID_PARMS, if an invalid color register was requested, or AX = ERROR_VIO_INVALID_LENGTH, if too many registers were requested.

Set Color Lookup Table

This function loads the definitions of the colors from the Color Lookup Table.

Subfunction Number: 107H (263)

Parameter Block Format

Length of parameter block in bytes (=12) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical hardware is updated Off = Update only the environment buffer. On = Update the environment buffer and the hardware state. Bits 1-15 are reserved and must be Off.	
Far address of Color Lookup Table. The Table format is device-dependent. The VGA format has three bytes containing the red, green and blue indices, respectively, for each color being set.	DWORD
Index of first table entry to set.	WORD
Number of table entries to set.	WORD

Remarks: This function returns with AX=0 if it can successfully set all of the registers in the Color Lookup Table. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS, if an invalid color register was requested, or AX = ERROR_VIO_INVALID_LENGTH, if too many registers were requested.

Query Cursor Info

This function returns all of the information related to the cursor.

Subfunction Number: 108H (264)

Parameter Block Format

Length of parameter block in bytes (=16) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical hardware is to be read. Off = Return data from the environment buffer only. On = Read the hardware to update the environment buffer before returning the requested data. The remaining flags select the information to be returned: Bit 1 selects cursor position. Bit 2 selects cursor type. Bits 3-15 are reserved and must be Off.	
Row (where 0 is the top row)	WORD
Column (where 0 is the left column)	WORD
Top cursor scan line. If n scan lines, 0 is top scan line and n-1 is bottom scan line.)	WORD
Bottom cursor scan line. If n scan lines, 0 is top scan line and n-1 is bottom scan line.)	WORD
Cursor width (in columns, if text mode; in pels, if graphics mode)	WORD
Cursor attribute (-1 = hidden, if text mode; other values = color attribute, if graphics mode)	WORD

Remarks: This function returns with AX=0, if it can successfully get all of the cursor information requested. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

Set Cursor Info

This function sets all of the information related to the cursor.

Subfunction Number: 109H (265)

Parameter Block Format

Length of parameter block in bytes (=16) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical hardware is updated Off = Update only the environment buffer. On = Update the environment buffer and the hardware state. The remaining flags select the options to be set: Bit 1 selects cursor position. Bit 2 selects cursor type. Bits 3-15 are reserved and must be Off.	
Row (where 0 is the top row)	WORD
Column (where 0 is the left column)	WORD
Top cursor scan line. If n scan lines, 0 is top scan line and n-1 is bottom scan line).	WORD
Bottom cursor scan line. If n scan lines, 0 is top scan line and n-1 is bottom scan line).	WORD
Cursor width (in columns, if text mode; in pels, if graphics mode)	WORD
Cursor attribute (-1 = hidden, if text mode; other values = color attribute, if graphics mode)	WORD

Remarks: This function returns with AX=0, if it can successfully set all of the cursor information requested. Otherwise, it returns with AX equal to:

ERROR_VIO_MODE, if it cannot support the function in the current mode
 ERROR_VIO_ROW, if the row number is out of range
 ERROR_VIO_COL, if the column number is out of range.

Query Font

This function returns the current active font or a selected font for the current code page. The format of the font definition is determined by the type of adapter.

Subfunction Number: 10AH (266)

Parameter Block Format

Length of parameter block in bytes (=14) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical hardware is to be read. Off = Return data from the environment buffer only. On = Read the hardware to update the environment buffer before returning the requested data.	
Bit 1 indicates whether a specific font is to be returned instead of the current font. Off = Return the current font. On = Return the selected font for the current code page. Setting this flag indicates that the pel columns and rows are used as input to select the font.	
Bits 2-15 are reserved and must be Off.	
Far address of the font buffer.	DWORD
Data area in which the font definition is returned.	
Length of data area in which font table is returned.	WORD
Pel columns.	WORD
Pel rows.	WORD

Remarks: If the Length is specified as 0, no font is returned. Instead, the Length field returns the size needed to hold the font. Query Font returns with AX=0, if it can successfully read the font. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

Set Font

This function sends a user font definition to the device handler. If the font is appropriate for the current mode, it is loaded into the adapter. If not, it is saved for possible use on subsequent calls to SetMode. The format of the font definition is determined by the type of adapter.

Subfunction Number: 10BH (267)

Parameter Block Format

Length of parameter block in bytes (=14) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical hardware is updated	
Off = Update only the environment buffer.	
On = Update the environment buffer and the hardware state.	
Bits 1-15 are reserved and must be Off.	
Far address of the font buffer containing the font set in compact form.	DWORD
Length of data area containing the font table to be set.	WORD
Pel columns	WORD
Pel rows	WORD

Remarks: This function returns with AX=0, if it can successfully load the font. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

Query Mode

This function returns all of the information pertaining to the current video mode.

Subfunction Number: 10CH (268)

Parameter Block Format

Length of parameter block in bytes (=8) passed on input WORD

Flags WORD

Bit 0 indicates whether the physical hardware is to be read.

Off = Return data from the environment buffer only.

On = Read the hardware to update the environment buffer before
returning the requested data.

Bits 1-15 are reserved and must be Off.

Far Address of the Mode data structure defined by VioGetMode. DWORD

Remarks: If the Length specified in the Config Data is larger than the maximum possible length, or if the Length is specified as 2, it is replaced by the largest valid length. This function returns with AX=ERROR_VIO_INVALID_LENGTH, only if the Length specified is less than 2.

Set Mode

This function sets the video mode of the video adapter. For text modes, it considers not only what display characteristics it can support, but also what ROM, code page, and user defined fonts it has available.

Subfunction Number: 10DH (269)

Parameter Block Format

Length of the data structure in bytes, including Length itself (=8). WORD

Flags WORD

Bit 0 indicates whether the physical hardware is updated.

Off = Update only the environment buffer.

On = Update the environment buffer and the hardware state.

Bit 1 indicates whether the mode is changed or only validated.

Off = Perform normal mode setting.

On = Perform only mode validation.

Bits 1-15 are reserved and must be Off.

Far Address of the Mode data structure defined by VioSetMode. DWORD

Remarks: This function must validate the mode data without using the environment buffer, because it might not have been initialized, or might not be valid for this device. This function implicitly initializes the Environment Buffer, if it has not already been done. Set Mode returns with AX=0, if it can set the requested mode. Otherwise, it returns with AX = ERROR_VIO_MODE.

Query Palette Registers

This function queries the relationship between the text attributes and the color registers.

Subfunction Number: 10EH (270)

Parameter Block Format

Length of parameter block in bytes (=12) passed on input WORD

Flags WORD

Bit 0 indicates whether the physical hardware is to be read.

Off = Return data from the environment buffer only.

On = Read the hardware to update the environment buffer before returning the requested data.

Bits 1-15 are reserved and must be Off.

Far address of the palette buffer. DWORD

Data area where a 1-WORD entry for each register containing its color value is returned.

Index of first palette register to get. WORD

Number of registers to return. WORD

Remarks: This function returns with AX=0, if it can successfully get all of the requested palette registers. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS, if an invalid color register was requested, or AX = ERROR_VIO_INVALID_LENGTH, if too many registers were requested.

Set Palette Registers

This function defines the relationship between the text attributes and the color registers.

Subfunction Number: 10FH (271)

Parameter Block Format

Length of parameter block in bytes (=12) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical hardware is updated.	
Off = Update only the environment buffer.	
On = Update the environment buffer and the hardware state.	
Bits 1-15 are reserved and must be Off.	
Far address of palette register buffer.	DWORD
Data area with 1-WORD, containing the color value for each register set.	
Index of first palette register to set.	WORD
Number of registers to set.	WORD

Remarks: This function returns with AX=0, if it can successfully set all of the requested palette registers. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS, if an invalid color register was requested, or AX = ERROR_VIO_INVALID_LENGTH, if too many registers were requested.

Query Physical Buffer

This function returns an LDT selector which can be used to access the physical video buffer. The current physical video buffer is returned unless a specific address range is requested.

Subfunction Number: 110H (272)

Parameter Block Format

Length of parameter block in bytes (=12) passed on input	WORD
Flags. Must be zero.	WORD
Far Address of the Query Physical Buffer data structure defined by VioGetPhysBuf.	DWORD

Remarks: If the physical display buffer address and length passed on input are 0, this subfunction returns an LDT selector, which corresponds to the current mode.

A physical Video device driver must provide Read/Write access to the physical address range where the physical display buffer is located. The physical device driver must provide Read-only access to the physical address range where the ROM fonts are located. If the physical address passed on input is not within the physical display buffer or ROM font ranges, an error is returned.

Query Physical Buffer returns with AX=0, if it can successfully allocate the LDT selector. Otherwise, it returns with AX set by DevHlp, PhysToUVirt, or AX = ERROR_VIO_INVALID_PARMS, if the requested buffer resides outside the valid range for the device.

Free Physical Buffer

This function deallocates an LDT selector that was acquired by calling the Query Physical Buffer routine.

Subfunction Number: 111H (273)

Parameter Block Format

Length of parameter block in bytes (=6) passed on input	WORD
Flags. Must be zero.	WORD
LDT Selector	WORD

Remarks: This function always returns with AX=0.

Query Variable Info

This function reads various minor features of the video adapter including the blink state, border color, underscore line, and scrollable rectangle of the screen.

Subfunction Number: 112H (274)

Parameter Block Format

Length of parameter block in bytes (=26) passed on input WORD

Flags WORD

Bit 0 indicates whether the physical hardware is to be read.

Off = Return data from the environment buffer only.

On = Read the hardware to update the environment buffer before returning the requested data.

The remaining flags select the information to be returned:

Bit 1 selects blink versus background color.

Bit 2 selects overscan (border) color.

Bit 3 selects scan line for underscore.

Bit 4 selects video enable.

Bit 5 selects the display mask.

Bit 6 selects code page.

Bit 7 forces a code page set (used with 6).

Bit 8 gets the scrollable rectangle.

Bits 9-15 are reserved and must be Off.

Blink versus background intensity: WORD

0 = blink

1 = background intensity.

Overscan (border) color. WORD

Scan line for underscore (0-31). 32 = no underscore. WORD

Video enable: WORD

0 = Off

1 = On

Off means off until the physical device driver is told to turn it back on. If the video signal is turned off and then the mode is set, the signal must remain off.

Display mask DWORD

bit 0 = plane 0

```

•   •
•   •
•   •

```

bit 31 = plane 31

bit state = 0, plane disabled for display.

bit state = 1, plane enabled for display.

(Planes disabled for display result in 0 to palette.)

Code page.	WORD
Scrollable Rectangle - Left.	WORD
Scrollable Rectangle - Top.	WORD
Scrollable Rectangle - Right.	WORD
Scrollable Rectangle - Bottom.	WORD

The scrollable rectangle fields indicate the area of the screen that can scroll during scroll and write TTY operations.

Screen Rows:	The number of text rows in the current mode.	WORD
Screen Columns:	The number of text columns in the current mode.	WORD

Remarks: Query Variable Info returns with AX=0, if it can successfully get the selected variable information. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

Set Variable Info

This function sets the minor features of the video adapter including the blink state, border color, underscore line, and scrollable rectangle of the screen.

Subfunction Number: 113H (275)

Parameter Block Format

Length of parameter block in bytes (=26) passed on input WORD

Flags WORD

Bit 0 indicates whether the physical hardware is updated.

Off = Update only the environment buffer

On = Update the environment buffer and the hardware state.

The remaining flags select the options to be set:

Bit 1 selects blink versus background color.

Bit 2 selects overscan (border) color.

Bit 3 selects scan line for underscore.

Bit 4 selects video enable.

Bit 5 selects the display mask.

Bit 6 selects code page.

Bit 7 forces a code page set (used with 6).

Bit 8 gets the scrollable rectangle.

Bits 9-15 are reserved and must be Off.

Blink versus background intensity: WORD

0 = blink

1 = background intensity.

Overscan (border) color. WORD

Scan line for underscore (0-31). 32 = no underscore. WORD

Video enable: WORD

0 = Off

1 = On

Off means off until the physical device driver is told to turn it back on. If the video signal is turned off and then the mode is set, the signal must remain off.

Display mask DWORD

bit 0 = plane 0

```

•   •
•   •
•   •

```

bit 31 = plane 31

bit state = 0, plane disabled for display.

bit state = 1, plane enabled for display.

(Planes disabled for display result in 0 to palette.)

Code page.	WORD
Scrollable Rectangle - Left.	WORD
Scrollable Rectangle - Top.	WORD
Scrollable Rectangle - Right.	WORD
Scrollable Rectangle - Bottom.	WORD

The scrollable rectangle fields indicate the area of the screen that can scroll during scroll and write TTY operations.

Screen Rows:	Number of text rows in current mode. Reserved (0).	WORD
Screen Columns:	Number of text columns in current mode. Reserved (0).	WORD

Remarks: There are two types of code page sets. The first code page set allows a code page to be set while the mode of the display (as used by “Set Mode” on page 14-24) remains the same. The other type of code page set causes a change in the display mode. This occurs when switching between DBCS and non-DBCS code pages.

If bit 6 of the flags WORD is set, and bit 7 is clear, the code page is set when the adapter can use the code page without changing from DBCS mode to SBCS mode, or vice versa. The mode should be changed, if both bits 6 and 7 of the flags WORD are set, and the code page is used when the mode is changed from SBCS mode to DBCS mode. Bit 7 is ignored, if bit 6 is not set. This only applies to text modes. Graphics modes are not set to text modes by forcing a code page.

The BVH need not support any scrollable region other than the entire display area. The adapter may support any scrollable rectangle up to the size of the entire screen. All coordinates are in text display cells. This scrollable rectangle data is undefined for graphics modes.

Set Variable Info returns with AX=0, if it can successfully set the selected variable information. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

Terminate Environment

This function is used to notify the BVH that the environment is about to be freed so that any required cleanup may be performed by the BVH. If no Terminate Environment processing is required, this function can be omitted. PMERR_DEV_FUNC_NOT_INSTALLED is then returned in AX, but ignored by the video subsystem.

Subfunction Number: 114H (276)

Parameter Block Format

Length of parameter block in bytes (=6) passed on input	WORD
Flags. Reserved, must be Off.	WORD
Logical Video Buffer selector. A huge selector.	WORD

Print Screen

This function causes the contents of the current screen to be written to the printer handle provided. BVS provides a default routine which provides the same level of support as previous versions, if the vector is not replaced.

Subfunction Number: 115H (277)

Parameter Block Format

Length of parameter block in bytes (=8) passed on input WORD

Flags WORD

Bit 0 indicates whether the physical video buffer should be printed.

Off = Print only the contents of the logical video buffer.

On = Print the contents of the physical video buffer, if appropriate.

Bits 1-15 are reserved and must be Off.

Logical Video Buffer selector. A huge selector. WORD

Print Device Handle. File handle of the print device to be used. WORD

Write TTY

This function performs the functions of the call to VioWrtTTY. BVS provides a default routine which provides the same level of support as previous versions, if the vector is not replaced.

Subfunction Number: 116H (278)

Parameter Block Format

Length of parameter block in bytes (=14) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical video buffer needs to be updated. Off = The physical video buffer must not be updated. On = The physical video buffer must be updated.	
Bit 1 indicates whether the logical video buffer needs to be updated. Off = The logical video buffer can optionally be updated. On = The logical video buffer must be updated.	
Bit 2 indicates whether ANSI is active. Off = ANSI is not active. Escape sequences should be considered as text data. On = ANSI is active. Escape sequences must be handled locally or passed to the default routine in BVS through device chaining.	
Bit 3 indicates whether Ctrl-PrtSc is active. Off = Ctrl_PrtSc is not active. On = Ctrl_PrtSc is active. Characters need to be echoed to the printer locally or by the default routine in BVS through device chaining.	
Bits 4-15 are reserved and must be Off.	
Logical Video Buffer selector. A huge selector.	WORD
Far Address of character string to be written.	DWORD
Length of Character string to be written.	WORD
Print Device Handle. The file handle of the print device used for Ctrl-PrtSc.	WORD

Query LVB Info

This function returns information associated with the LVB such as the allocation size and default attribute for a specified LVB.

Subfunction Number: 117H (279)

Parameter Block Format

Length of parameter block in bytes (=20) passed on input	WORD
Flags	WORD
Bit 0 indicates whether the physical hardware is to be read. Off = Read from the environment buffer. On = Read from the current state of the hardware state. Bits 1-15 are reserved and must be Off.	
Format ID for LVB. If this and the attribute count are both 0, the current mode values are used.	BYTE
Attribute Count for the LVB.	BYTE
LVB Width in cells.	WORD
LVB Height in cells.	WORD
Allocation size of the LVB is returned here.	DWORD
Size of the default attribute return buffer.	WORD
Pointer to the default attribute return buffer (passed). The default attribute is returned, if the buffer is large enough. If this value is 0, the attribute is not returned.	DWORD

Remarks: This function returns with AX=0, if it can successfully calculate the LVB size and return the attribute information. Otherwise, it returns with AX = ERROR_VIO_INVALID_PARMS.

Level 0 Physical Device Driver Interfaces

The strategy portion of the Level 0 physical device drivers that can be a component of any video device handler, is called to handle I/O requests through a request packet interface with the OS/2 kernel. The strategy routine executes at task-time as a result of an application VIO request. The strategy routine is called with ES:BX pointing to the request packet (the pointer is valid in both a DOS Session and an OS/2 session). Only three command codes (passed in the request packet) are required to be supported by a video device driver. For any other command code that the physical device driver does not support, the physical Video device driver should return Unsupported Command and Done in the request packet status field.

The following are the physical device driver commands that are supported by the physical Video device driver (the command codes are in parentheses):

INIT (00H) - Initialize the Device

On entry, the request block contains the following fields as inputs to the physical Video device driver:

- Pointer to the DevHlp entry point
- Pointer to the INIT arguments.

On exit, the physical Video device driver sets the first pointer to the offsets of the code and data segments, to release code and data needed only by the initialization routine. The second pointer is set to 0. If initialization is successful, the request packet status field is set to indicate No Error and done, otherwise, the status is set to General Failure.

The physical Video device driver can perform the following initialization:

- Obtain the DevHlp address from the request packet
- Verify that the display adapter and display, which it supports are present. If not, it can fail initialization.

OPEN (0DH) - Open the Device

This service routine does nothing but return with No Error status.

GENERIC IOCTL (10H) - Send I/O Requests to the Device

On entry, the request packet has the IOCTL category code and function code set. The parameter buffer and the data buffer addresses are set as virtual addresses. The physical Video device driver performs the physical device driver initialization, when requested.

Physical Device Driver Initialization

The IOCTL described below supports the same class of functions as the Presentation Manager dynalink enable entry point. It is called to fill in the Call Vector Table for the BVH. The call returns an error if the physical device driver detects that the adapter is not present.

Category 3: Function 73H

Purpose: To initialize the Call Vector Table.

Parameter Block Format: None.

Data Packet Format

DWORD	Subfunction
DWORD	Parameter1
DWORD	Parameter2
WORD	ReturnCode

Subfunction: The Presentation Manager enable subfunction number. Its value must be 1 to invoke the Presentation Manager Fill Logical Device Block subfunction. All pieces of the BVH must support this function. Pieces of the BVH that do not support the Presentation Manager interface do not need to support the other functions. Any function not supported should return PMERR_DEV_FUNC_NOT_INSTALLED.

Parameter1 for Subfunction = 1: Far pointer to this structure:

DWORD	Engine Version
DWORD	Count of Table Functions

Engine Version Version of the Presentation Manager Graphics Engine

Count of Table Functions Number of entries in the passed dispatch table. The physical device driver can only write this many entries into the table.

Parameter2 for Subfunction = 1: Far pointer to this structure:

DWORD	Flags Pointer
DWORD	Call Vector Table

Flags Pointer Pointer to where the flags controlling calls to the Fill Physical Device Block function go.

Call Vector Table Pointer to the default dispatch table containing the addresses of the default handler functions and the functions supported by this component. Each entry in the Call Vector Table is the far address of a Video Device Handler function, which must be callable from Ring 3. The address of the *n*th BVH function is the *n*th DWORD in the table, beginning with Function 0. Refer to the DLL Initialization function for a description of the function numbers.

Remarks: This function is supported by the Video device drivers, and is only called once for each adapter to be supported by the BVH. A video device handler should determine if the display adapter, and that which it supports is present. If not present, this function should return an error. Every part of a BVH must successfully initialize the Call Vector Table for that device to be used by OS/2 2.0.

EGA.SYS and INT 2F Screen Switch Notification

For some DOS EGA applications, OS/2 2.0 is not able to switch from a DOS Session to an OS/2 session, and then back again. Upon return to the DOS application, the screen will be incorrect. The DOS EGA applications that do not run successfully are:

- Applications that download fonts into a character generator block other than Block 0. Character Generator Block 0 is supported.
- Graphic mode applications that use more than one display page.
- Advanced graphics mode applications that write directly to the registers on the EGA adapter.

To supplement OS/2 screen switching support, a DOS application can be written to use the EGA Register Interface. Alternatively, a DOS application can be notified on a screen switch through Multiplex Interrupt 2FH, AH=40H. These two mechanisms are described in the following sections.

Note: On an IBM PS/2 system, the registers on the adapter are both readable and writable. For these configurations, OS/2 2.0 reads and saves the registers on a screen switch away from a DOS Session, and restores the registers upon return to a DOS Session.

For configurations including the IBM PS/2 Display Adapter 8514/A when the 8514/A display adapter is in an advanced function mode, the OS/2 operating system does not save the physical display buffer, when switching away from a DOS Session. Therefore, end users are cautioned to complete any 8514/A advanced function mode application before switching to OS/2 mode.

EGA.SYS Device Driver

EGA.SYS is a physical device driver that provides support for the EGA Register Interface in a DOS Session. To support advanced graphics modes D, E, F, and 10 in a DOS Session, the Mouse Pointer Draw device driver must save or restore the EGA registers. Because the EGA registers are not readable, this can be done only if the application assists in setting the registers initially. Rather than performing I/O directly to the registers on the adapter, the application sets the registers through the EGA Register Interface.

EGA Register Interface

The EGA Register Interface is a library of ten functions supported for a DOS Session, advanced graphics applications (modes D, E, F, and 10). These functions do the following:

- Read from, or write to, one or more of the EGA Write-only registers
- Define default values for the EGA Write-only registers, reset the EGA registers to these default values, or return the default values
- Check whether the EGA Register Interface is present and, if so, return its version number.

When the application uses the EGA Register Interface, OS/2 2.0 maintains a backup copy, or *shadows* how the EGA registers are set. Then, if the operator switches away from (and later returns to) the application, the registers are restored properly. It is not necessary to use the EGA Register Interface to set the mode, color palette, or palette registers. Instead, use ROM BIOS function INT 10H with AH = 00H, 0BH, or 10H, respectively.

Calling The EGA Register Interface: To call EGA Register Interface functions from an assembly language program, the following actions must be performed:

1. Load the registers with the required parameter values
2. Execute software interrupt 10H.

Values returned by the EGA Register Interface functions are placed in registers.

EGA Register Interface Restrictions: A list of areas where restrictions apply for the EGA Register Interface are shown below:

- Functions not supported
- Attribute Controller registers
- Sequencer Memory Mode register
- Input Status registers
- Graphics Controller Miscellaneous register

Functions Not Supported: Multiple display pages in graphics modes are not supported. Fonts can be loaded (by using ROM BIOS INT 10H with AH = 11H) only into Character Generator Block 0.

Attribute Controller Registers: Before your application program uses the Attribute Controller registers (I/O address 3C0H) in an extended interrupt 10H call, it must set the *flip-flop* that selects the address or data register, so that it selects the address register (by doing an input from I/O port 3BAH or 3DAH). The flip-flop is always reset to this state upon return from the extended INT 10H call. Interrupt routines that access the attribute chip must also leave the flip-flop set to the address register upon return from the interrupt.

Note: If the application program sets the flip-flop so that it selects the Data register, and expects the flip-flop to remain in this state, the application must disable interrupts between the time it sets the flip-flop to the Data register state, and the last time the flip-flop is assumed to be in this state.

Sequencer Memory Mode Register: When the Sequencer Memory Mode register (I/O address 3C5H, data register 4) is accessed, the sequencer produces a *fault* on the CAS lines that can cause problems with video random access memory. As a result, the application cannot use the EGA Register Interface to read from, or write to, this register. Instead, use the following procedure to safely alter this register:

1. Disable interrupts
2. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 0
3. Read/modify/write the Sequencer Memory Mode register
4. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 1
5. Enable interrupts.

Input Status Registers: The application cannot use the EGA Register Interface to read Input Status registers 0 (I/O address 3C2H), and 1 (I/O address 3BAH or 3DAH). If the program must read these registers, it should do so directly.

Graphics Controller Miscellaneous Register: When the Graphics Controller Miscellaneous register (I/O address 3CFH, data register 6) is accessed, a *glitch* on the CAS lines occurs that can cause problems with video random access memory. As a result, the application should not use the EGA Register Interface to read from or write to this register.

EGA Register Interface Function F6 does not alter the state of the Graphics Controller Miscellaneous register. Instead, use the following procedure to safely alter this register:

1. Disable interrupts
2. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 0
3. Read/modify/write the Graphics Controller Miscellaneous register
4. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 1
5. Enable interrupts.

EGA Register Interface Functions: This section describes each EGA Register Interface function in detail. The following list shows these functions by function number (hex):

F0	Read One Register
F1	Write One Register
F2	Read Register Range
F3	Write Register Range
F4	Read Register Set
F5	Write Register Set
F6	Revert to Default Registers
F7	Define Default Register Table
F8	Read Default Register Table
FA	Interrogate Driver.

Note: Function F9H, and Functions FBH through FFH are reserved.

Each function description includes:

- The parameters required to make the call (input) and the expected return values (output)
- Any special considerations regarding the function.

If the function description does not specify an input for a parameter, it is not necessary to supply a value for that parameter before making the call. If the function description does not specify an output value for a parameter, the parameter's value is the same before and after the call.

Note: The EGA Register Interface does not check input values, so be sure that the values loaded into the registers before making a call are correct.

Function F0 – Read One Register

This function reads data from a specified register on the EGA.

Input

AH = F0H

BX = Pointer for:

Pointer/data chips

BH = 0

BL = Pointer

Single registers

BX ignored.

DX = Port number:

Pointer/data chips

0H: CRT Controller (3?4H)

8H: Sequencer (3C4H)

10H: Graphics Controller (3CEH)

18H: Attribute Controller (3C0H)

Single registers

20H: Miscellaneous Output register (3C2H)

28H: Feature Control register (3?AH)

30H: Graphics 1 Position register (3CCH)

38H: Graphics 2 Position register (3CAH)

? = B for monochrome modes, or D for color modes

Output

AX: Restored

BH: Restored

BL: Data

DX: Restored

All other registers restored.

Examples: The following example saves the contents of the Sequencer Map Mask register in *myvalue*:

```
myvalue db ?
mov ah, 0f0h      ; f0 = read one register
mov bx, 0002h     ; bh = 0 / bl = map mask index
mov dx, 0008h     ; dx = sequencer
int 10h           ; get it!
mov myvalue, bl   ; save it!
```

The example below saves the contents of the Miscellaneous Output register in *myvalue*:

```
myvalue db ?
mov ah, 0f0h      ; f0 = read one register
mov dx, 0020h     ; dx = miscellaneous output register
int 10h           ; get it!
mov myvalue, bl   ; save it!
```

Function F1 – Write One Register

This function writes data to a specified register on the EGA. When an application program returns from a call to Function F1, the contents of registers BH and DX are not restored. The program must save and restore these registers, if desired.

Input

AH = F1H

BL = Pointer for pointer/data chips. Data for single registers.

BH = Data for pointer/data chips. Ignored for single registers.

DX = Port number:

Pointer/data chips

0H: CRT Controller (3?4H)

8H: Sequencer (3C4H)

10H: Graphics Controller (3CEH)

18H: Attribute Controller (3C0H)

Single registers

20H: Miscellaneous Output register (3C2H)

28H: Feature Control register (3?AH)

30H: Graphics 1 Position register (3CCH)

38H: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

Output

AX: Restored

BL: Restored

BH: Not restored

DX: Not restored

All other registers restored.

Examples: The following example writes the contents of *myvalue* into the CRT Controller Cursor Start register:

```
myvalue db 3h
        mov ah, 0f1h      ; f1 = write one register
        mov bh, myvalue   ; bh = data from myvalue
        mov bl, 000ah     ; bl = cursor start index
        mov dx, 0000h     ; dx = crt controller
        int 10h           ; write it!
```

The example below writes the contents of *myvalue* into the Feature Control register:

```
myvalue db 2h
        mov ah, 0f1h      ; f1 = write one register
        mov bl, myvalue   ; bl = data from myvalue
        mov dx, 0028h     ; dx = feature control register
        int 10h           ; write it!
```

Function F2 – Read Register Range

This function reads data from a specified range of registers on the EGA. A range of registers is defined to be several registers on a single chip that have consecutive indexes. This function is applicable for pointer/data chips.

Input

AH = F2H

CH = Starting pointer value

CL = Number of registers (must be > 1)

DX = Port number:

Pointer/data chips

0h: CRT Controller (3?4H)

8h: Sequencer (3C4H)

10h: Graphics Controller (3CEH)

18h: Attribute Controller (3C0H)

? = B for monochrome modes, or D for color modes

ES:BX = Points to table of one-byte entries (length = value in CL). On return, each entry is set to the contents of the corresponding register.

Output

AX: Restored

BX: Restored

CX: Not restored

DX: Restored

ES: Restored

All other registers restored.

Example: The following example saves the contents of the Attribute Controller Palette registers in *paltable*:

```
paltable db 16 dup (?)
        mov ax, ds          ; assume paltable in data segment
        mov es, ax          ; es = data segment
        mov bx, offset paltable ; es:bx = paltable address
        mov ah, 0f2h        ; f2 = read register range
        mov cx, 0010h       ; ch = start index of 0
                                ; cl = 16 registers to read
        mov dx, 0018h       ; dx = attribute controller
        int 10h             ; read them!
```

Function F3 – Write Register Range

This function writes data to a specified range of registers on the EGA. A range of registers is defined to be several registers on a single chip that have consecutive indexes. This function is applicable for the pointer/data chips.

Input

AH = F3H

CH = Starting pointer value

CL = Number of registers (must be > 1)

DX = Port number:

Pointer/data chips

0H: CRT Controller (3?4H)

8H: Sequencer (3C4H)

10H: Graphics Controller (3CEH)

18H: Attribute Controller (3C0H)

? = B for monochrome modes, or D for color modes

ES:BX = Points to table of one-byte entries (length = value in CL). Each entry contains the value to be written to the corresponding register.

Output

AX: Restored

BX: Not restored

CX: Not restored

DX: Not restored

ES: Restored.

All other registers restored.

Example: The following example writes the contents of *cursloc* into the CRT Controller Cursor Location High and Cursor Location Low registers.

```
cursloc db 01h, 00h          ; cursor at page offset 0100h
        mov ax, ds            ; assume cursloc in data segment
        mov es, ax            ; es = data segment
        mov bx, offset cursloc ; es:bx = cursloc address
        mov ah, 0f3h          ; f3 = write register range
        mov cx, 0e02h         ; ch = start index of 14
                                ; cl = 2 registers to write
        mov dx, 0000h         ; dx = crt controller
        int 10h               ; write them!
```

Function F4 – Read Register Set

This function reads data from a set of registers on the EGA. A set of registers is defined to be several registers that might have consecutive indexes, and that might not be on the same chip.

Input

AH = F4H

CX = Number of registers (must be > 1)

ES:BX = Points to table of records with each entry in this format:

Bytes 1-2: Port number:

Pointer/data chips

0H: CRT Controller (3?4H)

8H: Sequencer (3C4H)

10H: Graphics Controller (3CEH)

18H: Attribute Controller (3C0H)

Single registers

20H: Miscellaneous Output register (3C2H)

28H: Feature Control register (3?AH)

30H: Graphics 1 Position register (3CCH)

38H: Graphics 2 Position register (3CAH)

? = B for monochrome modes, or D for color modes

Byte 3: Pointer value (0 for single registers)

Byte 4: EGA Register Interface fills in data read from register specified in bytes 1-3.

Output

AX: Restored

BX: Restored

CX: Not restored

ES: Restored.

All other registers restored.

Example: The following example saves the contents of the Miscellaneous Output register, Sequencer Memory Mode register, and CRT Controller Mode Control register in *results*:

```

outvals dw 0020h          ; miscellaneous output register
        db 0              ; 0 for single registers
        db ?             ; returned value
        dw 0008h          ; sequencer
        db 04h           ; memory mode register index
        db ?             ; returned value
        dw 0000h          ; crt controller
        db 17h           ; mode control register index
        db ?             ; returned value

results db 3 dup (?)
        mov ax, ds        ; assume outvals in data segment
        mov es, ax        ; es = data segment
        mov bx, offset outvals ; es:bx = outvals address
        mov ah, 0f4h      ; f4 = read register set
        mov cx, 3         ; number of entries in outvals
        int 10h           ; get values into outvals
        mov si, 3         ; move the returned values from
        add si, offset outvals ; outvals
        mov di, offset results ; to results
        mov cx, 3         ; 3 values to move

loop:   mov al, [si]       ; move one value from
        mov [di], al      ; outvals to results
        add si, 4         ; skip to next source byte
        inc di            ; point to next destination byte
        loop loop

```

Function F5 – Write Register Set

This function writes data to a set of registers on the EGA. A set of registers is defined to be several registers that might have consecutive indexes, and that might be on the same chip.

Input

AH = F5H

CX = Number of registers (must be > 1)

ES:BX = Points to table of values with each entry in this format:

Bytes 1-2: Port number:

Pointer/data chips

0H: CRT Controller (3?4H)

8H: Sequencer (3C4H)

10H: Graphics Controller (3CEH)

18H: Attribute Controller (3C0H)

Single registers

20H: Miscellaneous Output register (3C2H)

28H: Feature Control register (3?AH)

30H: Graphics 1 Position register (3CCH)

38H: Graphics 2 Position register (3CAH)

? = B for monochrome modes, or D for color modes

Byte 3: Pointer value (0 for single registers)

Byte 4: Data to be written to register specified in bytes 1-3.

Output

AX: Restored

BX: Restored

CX: Not restored

ES: Restored.

All other registers restored.

Example: The following example writes the contents of *outvals* to the Miscellaneous Output register, Sequencer Memory Mode register, and CRT Controller Mode Control register:

```
outvals dw 0020h      ; miscellaneous output register
         db 0          ; 0 for single registers
         db 0a7h       ; output value
         dw 0008h      ; sequencer
         db 04h        ; memory mode register index
         db 03h        ; output value
         dw 0000h      ; crt controller
         db 17h        ; mode control register index
         db 0a3h       ; output value
         mov ax, ds     ; assume outvals in data segment
         mov es, ax     ; es = data segment
         mov bx, offset outvals ; es:bx = outvals address
         mov ah, 0f5h   ; f5 = write register set
         mov cx, 3      ; number of entries in outvals
         int 10h        ; write the registers!
```

Function F6 – Revert to Default Registers

This function restores the default settings of any registers that the application program has changed through the EGA Register Interface. The default settings are defined in a call to Function F7 (described in the next section).

Input

AH = F6H

Output: All registers restored.

Example: The following example restores the default settings of the EGA registers:

```
mov  ah, 0f6h    ; f6 = revert to default registers
int  10h         ; do it now!
```


Function F7 – Define Default Register Table

This function defines a table containing default values for any pointer/data chip or single register. If default values are defined for a pointer/data chip, they must be defined for all registers within that chip.

Input

AH = F7H

DX = Port number:

Pointer/data chips

0H: CRT Controller (3?4H)

8H: Sequencer (3C4H)

10H: Graphics Controller (3CEH)

18H: Attribute Controller (3C0H)

Single registers

20H: Miscellaneous Output register (3C2H)

28H: Feature Control register (3?AH)

30H: Graphics 1 Position register (3CCH)

38H: Graphics 2 Position register (3CAH)

? = B for monochrome modes, or D for color modes

ES:BX = Points to table of one-byte entries. Each entry contains the default value for the corresponding register. The table must contain entries for all registers.

Output

AX: Restored

BX: Not restored

DX: Not restored

ES: Restored

All other registers restored.

Examples: The following example defines default values for the Attribute Controller:

```
attrdflt db 00h, 01h, 02h, 03h, 04h, 05h, 06h, 07h
          db 10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h
          db 08h, 00h, 0fh, 00h
mov ax, ds          ; assume attrdflt in data segment
mov es, ax          ; es = data segment
mov bx, offset attrdflt ; es:bx = attrdflt address
mov ah, 0f7h        ; f7 = define default register table
mov dx, 0018h       ; dx = attribute controller
int 10h             ; do it!
```

The example below defines a default value for the Feature Control register:

```
featdflt db 00h
mov ax, ds          ; assume featdflt in data segment
mov es, ax          ; es = data segment
mov bx, offset featdflt ; es:bx = featdflt address
mov ah, 0f7h        ; f7 = define default register table
mov dx, 0028h       ; dx = feature control register
int 10h             ; do it!
```

Function F8 – Read Default Register Table

This function reads the table containing default register values for any pointer/data chip or single register.

Input

AH = 0F8H

DX = Port number:

Pointer/data chips

0H: CRT Controller (3?4H)

8H: Sequencer (3C4H)

10H: Graphics Controller (3CEH)

18H: Attribute Controller (3C0H)

Single registers

20H: Miscellaneous Output register (3C2H)

28H: Feature Control register (3?AH)

30H: Graphics 1 Position register (3CCH)

38H: Graphics 2 Position register (3CAH)

? = B for monochrome modes, or D for color modes

ES:BX = Points to a table into which the default values are returned. The table must have room for the full set of values for the pointer/data chip or single register specified.

Output

AX: Restored

BX: Not restored

DX: Not restored

ES: Restored

All other registers restored.

Example: The following example reads the default values for the Miscellaneous Output register:

```
regdflt db
mov ax, ds
mov es, ax           ; es = data segment
mov bx, offset regdflt ; es:bx = regdflt address
mov ah, 0f8h         ; f8 = read default register table
mov dx, 0020h        ; dx = miscellaneous output register
int 10               ; do it!
```

The following example reads the default values for the CRT Controller register:

```
regdflt db 25 dup (?)
mov ax, ds
mov es, ax           ; es = data segment
mov bx, offset regdflt ; es:bx = regdflt address
mov ah, 0f8h         ; f8 = read default register table
mov dx, 0000h        ; dx = crt controller register
int 10               ; do it!
```

Function FA — Interrogate Driver

This function returns a value specifying whether the EGA.SYS device driver is present.

Input

AH = FAH

BX = 0

Output

AX: Restored

BX: 0, if EGA.SYS driver is not present

ES:BX: Pointer to EGA Register Interface version number, if present:

Byte 1: Major release number

Byte 2: Minor release number (in 1/100ths)

Example: The following example interrogates the driver and displays the results:

```
gotmsg db "EGA.SYS driver found", 0dh, 0ah, 24h
nopmsg db "EGA.SYS driver not found", 0dh, 0ah, 24h
revmsg db "revision $"
crlf db 0dh, 0ah, 24h
ten db 10

mov bx, 0 ; must be 0 for this call
mov ah, 0fah ; fa = interrogate driver
int 10h ; interrogate!
or bx, bx ; bx = 0 ?
jnz found ; branch if driver present
mov dx, offset nopmsg ; assume nopmsg in data segment
mov ah, 09h ; 9 = print string
int 21h ; output not found message
jmp continue ; that's all for now
found: mov dx, offset gotmsg ; assume gotmsg in data segment
mov ah, 09h ; 9 = print string
int 21h ; output found message
mov dx, offset revmsg ; assume revmsg in data segment
mov ah, 09h ; 9 = print string
int 21h ; output "revision "
mov dl, es:[bx] ; dl = major release number
add dl, "0" ; convert to ASCII
mov ah, 2 ; 2 = display character
int 21h ; output major release number
mov dl, "." ; dl = "."
mov ah, 2 ; 2 = display character
int 21h ; output a period
mov al, es:[bx+1] ; al = minor release number
xor ah, ah ; ah = 0
idiv ten ; al = 10ths, ah = 100ths
mov bx, ax ; save ax in bx
mov dl, al ; dl = 10ths
```

```
add dl, "0"      ; convert to ASCII
mov ah, 2        ; 2 = display character
int 21h          ; output minor release 10ths
mov dl, bh       ; dl = 100ths
add dl, "0"      ; convert to ASCII
mov ah, 2        ; 2 = display character
int 21h          ; output minor release 100ths
mov dx, offset crlf ; assume crlf in data segment
mov ah, 09h      ; 9 = print string
int 21h          ; output end of line
continue:        ; the end
```

INT 2F Screen Switch Notification

The OS/2 operating system issues a new Multiplex Interrupt (INT 2FH) to signal the following two events:

- Switching the DOS application to the background (AX=4001H)
- Switching the DOS application to the foreground (AX=4002H).

A DOS application that receives this signal must *hook* the Multiplex Interrupt vector. That is, when the application is started, it must save the current INT 2FH vector and set this vector to point to the application's interrupt handler.

When the notification is received, the application must save all registers, perform whatever processing is required, restore all registers, and issue the IRET instruction to return to the operating system. Only the following forms of processing are supported:

- Modifying application and/or video memory
- Issuing ROM BIOS video service calls (INT 10H)
- Issuing EGA Register Interface calls (INT 10H)
- Programming the EGA video card directly.

Note: When an application is being switched to the background, if the application's INT 2F handler uses the EGA Register Interface to save the EGA registers, these registers are restored automatically when the application is returned to the foreground.

An application can receive notification that it is being switched to the background at any time. Code sequences that are sensitive to interruption can be protected with CLI/STI instructions. At the point the switch notification occurs, the application (other than its INT 2FH handler) is frozen until it is returned to the foreground.

When an application's INT 2FH handler receives notification with a value in AH other than 40H, the application must issue the JMP FAR instruction to branch to the previous INT 2FH vector.

Using Extended Screen and Keyboard Control (ANSI.SYS, ANSICALL)

This section explains how to issue special control character sequences to:

- Control the position of the cursor
- Erase text from the screen
- Set the display mode
- Redefine the meaning of keyboard keys.

ANSI extended screen and keyboard control sequences are supported in the DOS Session by ANSI.SYS, an installable device driver. In the OS/2 session, these control sequences are supported by the ANSICALL component within OS/2 CHAR.DLL.

Note: In this section, unless otherwise specified, ANSI refers to both ANSI.SYS and ANSICALL.

Limitations/Restrictions

ANSI operates on a per-session basis. OS/2-mode ANSI is affected when keys are reassigned in a code page environment. ANSI does not provide code page support for key reassignment in a DOS Session.

Control Sequence Syntax

Each of the cursor control sequences is in the format:

ESC [parameters COMMAND

A cursor control sequence is defined as follows:

ESC	The 1-byte ASCII code for ESC (1BH). It is not the three characters ESC.
[The character [.
parameters	The numeric values specified for #. The # represents a numeric parameter, which is an integer value specified with ASCII characters. If a parameter value is not specified, or if a value of 0 is specified, the default value for the parameter is used.
COMMAND	An alphabetic string that represents the command. It is case specific.

Control Sequence

For example, ESC [2;10H could be created using BASIC as shown below. Notice that "CHR\$(27)" is ESC.

The IBM Personal Computer Basic Version 3.00 Copyright IBM Corp. 1981, 1982, 1983, 1984
xxxxx Bytes free

```
OK
open "sample" for output as 1
OK
print #1, CHR$(27);"[2;10H";"x row 2 col 10"
OK
close #1
OK
```

Cursor Control Sequences

The following tables contain the cursor control sequences used to control cursor positioning:

Cursor Position	Function
ESC [#;#H	Moves the cursor to the position specified by the parameters. The first parameter specifies the row number and the second parameter specifies the column number. The default value is 1. If no parameter is given, the cursor is moved to the home position.

This example copies the file SAMPLE from the previous example, to CON, which places the cursor on row 2, column 10 of the screen:

```
type sample
```

Cursor Up	Function
ESC [#A	Moves the cursor up one or more rows without changing the column position. The value of # determines the number of lines moved. The default value for # is 1. This sequence is ignored, if the cursor is already on the top line.

Cursor Down	Function
ESC [#B	Moves the cursor down one or more rows without changing the column position. The value of # determines the number of lines moved. The default value for # is 1. The sequence is ignored, if the cursor is already on the bottom line.

Cursor Forward	Function
ESC [#C	Moves the cursor forward one or more columns without changing the row position. The value of # determines the number of columns moved. The default value for # is 1. This sequence is ignored, if the cursor is already in the rightmost column.

Cursor Backward	Function
ESC [#D	Moves the cursor back one or more columns without changing the row position. The value of # determines the number of columns moved. The default value for # is 1. This sequence is ignored, if the cursor is already in the leftmost column.

Horizontal and Vertical Position	Function
ESC [##;#t	Moves the cursor to the position specified by the parameters. The first parameter specifies the line number and the second parameter specifies the column number. The default value is 1. If no parameter is given, the cursor is moved to the home position.

Cursor Position Report	Function
ESC [##;#R	The cursor sequence report reports the current cursor position through the standard input device. The first parameter specifies the current line, and the second parameter specifies the current column.

Device Status Report	Function
ESC [6n	The console driver gives a cursor position report sequence on receipt of device status report.

Note: Do not use the Device Status Report as part of a prompt.

This example tells ANSI to put the current cursor position (row and column) in STDIN. Then the program reads it from STDIN, and outputs to STDOUT.

```
PROGRAM dsr(INPUT,OUTPUT);
```

```
VAR
  f:FILE OF CHAR;
  key:CHAR;

FUNCTION inkey:CHAR;
  VAR
    ch:CHAR;
  BEGIN
    READ(f,ch);
    inkey:=ch
  END;

BEGIN
  ASSIGN(f,'user');
  RESET(f);
  WRITE(CHR(27),'[6n');
  key:=inkey;
  key:=inkey;
  key:=inkey;
  WRITE('row ',inkey,inkey,' column ');
  key:=inkey;
  WRITE(inkey,inkey)
END.
```

Save Cursor Position	Function
ESC [s	The current cursor position is saved. This cursor position can be restored with the restore cursor position sequence (see below).

Restore Cursor Position

Restore Cursor Position	Function
ESC [u	Restores the cursor to the value it had when the console driver received the save cursor position sequence.

Erasing

The following tables contain the control sequences used to erase text from the screen.

Erase In Display	Function
ESC [2J	Erases all of the screen and the cursor goes to the home position.

Erase In Line	Function
ESC [K	Erases from the cursor to the end of the line and includes the cursor position.

Controlling Display Mode

The following tables contain the control sequences used to set the mode of operation:

- Set Graphics Rendition (SGR)
- Set Mode (SM)
- Reset Mode (RM).

Set Graphics Rendition (SGR)	Function																																																		
ESC [# ; ... ; # m	Sets the character attribute specified by the parameters. All following characters have the attribute according to the parameters until the next occurrence of SGR.																																																		
	<table> <tr> <th>Parameter</th><th>Meaning</th></tr> <tr><td>0</td><td>All attributes off (normal white on black)</td></tr> <tr><td>1</td><td>Bold on (high intensity)</td></tr> <tr><td>4</td><td>Underscore on (mono-compatible modes)</td></tr> <tr><td>5</td><td>Blink on</td></tr> <tr><td>7</td><td>Reverse video on</td></tr> <tr><td>8</td><td>Canceled on (invisible)</td></tr> <tr><td>30</td><td>Black foreground</td></tr> <tr><td>31</td><td>Red foreground</td></tr> <tr><td>32</td><td>Green foreground</td></tr> <tr><td>33</td><td>Yellow foreground</td></tr> <tr><td>34</td><td>Blue foreground</td></tr> <tr><td>35</td><td>Magenta foreground</td></tr> <tr><td>36</td><td>Cyan foreground</td></tr> <tr><td>37</td><td>White foreground</td></tr> <tr><td>38</td><td>Reserved</td></tr> <tr><td>39</td><td>Reserved</td></tr> <tr><td>40</td><td>Black background</td></tr> <tr><td>41</td><td>Red background</td></tr> <tr><td>42</td><td>Green background</td></tr> <tr><td>43</td><td>Yellow background</td></tr> <tr><td>44</td><td>Blue background</td></tr> <tr><td>45</td><td>Magenta background</td></tr> <tr><td>46</td><td>Cyan background</td></tr> <tr><td>47</td><td>White background</td></tr> </table>	Parameter	Meaning	0	All attributes off (normal white on black)	1	Bold on (high intensity)	4	Underscore on (mono-compatible modes)	5	Blink on	7	Reverse video on	8	Canceled on (invisible)	30	Black foreground	31	Red foreground	32	Green foreground	33	Yellow foreground	34	Blue foreground	35	Magenta foreground	36	Cyan foreground	37	White foreground	38	Reserved	39	Reserved	40	Black background	41	Red background	42	Green background	43	Yellow background	44	Blue background	45	Magenta background	46	Cyan background	47	White background
Parameter	Meaning																																																		
0	All attributes off (normal white on black)																																																		
1	Bold on (high intensity)																																																		
4	Underscore on (mono-compatible modes)																																																		
5	Blink on																																																		
7	Reverse video on																																																		
8	Canceled on (invisible)																																																		
30	Black foreground																																																		
31	Red foreground																																																		
32	Green foreground																																																		
33	Yellow foreground																																																		
34	Blue foreground																																																		
35	Magenta foreground																																																		
36	Cyan foreground																																																		
37	White foreground																																																		
38	Reserved																																																		
39	Reserved																																																		
40	Black background																																																		
41	Red background																																																		
42	Green background																																																		
43	Yellow background																																																		
44	Blue background																																																		
45	Magenta background																																																		
46	Cyan background																																																		
47	White background																																																		

Set Mode (SM)	Function																		
ESC [=#h or ESC [=h or ESC [=0h or ESC [?7h	Invokes the screen width or type specified by the parameter.																		
	<table border="1"> <thead> <tr> <th>Parameter</th><th>Meaning</th></tr> </thead> <tbody> <tr> <td>0</td><td>40x25 black and white</td></tr> <tr> <td>1</td><td>40x25 color</td></tr> <tr> <td>2</td><td>80x25 black and white</td></tr> <tr> <td>3</td><td>80x25 color</td></tr> <tr> <td>4</td><td>320x200 color</td></tr> <tr> <td>5</td><td>320x200 black and white</td></tr> <tr> <td>6</td><td>640x200 black and white</td></tr> <tr> <td>7</td><td>Wrap at end of line. Typing past end-of-line results in new line.</td></tr> </tbody> </table>	Parameter	Meaning	0	40x25 black and white	1	40x25 color	2	80x25 black and white	3	80x25 color	4	320x200 color	5	320x200 black and white	6	640x200 black and white	7	Wrap at end of line. Typing past end-of-line results in new line.
Parameter	Meaning																		
0	40x25 black and white																		
1	40x25 color																		
2	80x25 black and white																		
3	80x25 color																		
4	320x200 color																		
5	320x200 black and white																		
6	640x200 black and white																		
7	Wrap at end of line. Typing past end-of-line results in new line.																		

Reset Mode (RM)	Function
ESC [=#I or ESC [=I or ESC [=OI or ESC [?7I	Parameters are the same as Set Mode (SM) except that parameter 7 resets wrap at end-of-line mode (characters past end-of-line are thrown away).

Keyboard Key Reassignment

The ANSI system can be used to reassign keys on the keyboard. When the application calls `KbdStringIn`, the reassigned key's ASCII code is converted to the specified string and is passed back to the calling application. For example, replace `a` with `q`, so that whenever the `a` key is pressed, a `q` is passed to the application that is reading input.

In OS/2 2.0, keyboard remapping can be done only from an application calling KbdStringIn. In DOS, keyboard remapping must be done from the command line.

Note: Keyboard reassignment only works with OS/2 applications that use the KbdStringIn call to get input.

OS/2 mode ANSI is affected when keys are reassigned in a code page environment. ANSI “remembers” the code page under which a key is reassigned. The keyboard subsystem checks for reassigned keys when the application calls the KbdStringIn function. When a reassigned key is detected, the ANSI support:

1. Checks to see what code page the requestor is running under
2. Looks internally to see if the key has been reassigned under that code page
3. If there is a key reassignment for that code page, gives the reassignment string
4. Otherwise, gives the original ASCII codes.

A maximum storage of 64KB can be allocated to OS/2 mode ANSI reassigned key definitions. The table shown below contains the control sequences used to redefine the meaning of keyboard keys:

The control sequence is:	Function
ESC [#;#...#p or ESC ["string"p or ESC [#;"string";#;#;"string";#p or any other combination of strings and decimal numbers.	The first ASCII code in the control sequence defines which code is being mapped. The remaining numbers define the sequence of ASCII codes generated when this key is intercepted. However, if the first code in the sequence is 0 (NULL), the first and second code make up an extended ASCII redefinition.

Keyboard Key Reassignment

To execute the examples below, either create a file that contains the following statements and then use the TYPE command to display the file that contains the statement, or execute the command at the OS/2 prompt:

- Assign the **A** and **a** key to the **Q** and **q** key, respectively.

- Assign the **Q** and **q** key to the **A** and **a** key, respectively:

Using a File:

```
ESC [65;81p      A becomes Q
ESC [97;113p     a becomes q
ESC [81;65p      Q becomes A
ESC [113;97p     q becomes a
```

At the OS/2 Prompt:

```
prompt $e[65;81p  A becomes Q
prompt $e[97;113p a becomes q
prompt $e[81;65p  Q becomes A
prompt $e[113;97p q becomes a
```

- Reassign the F10 key to a DIR command followed by a carriage return:

Using a File:

```
ESC [0;68;"dir";13p
```

At the OS/2 Prompt:

```
prompt $e[0;68;"dir";13p
```

The \$e is the prompt command characters for ESC. The 0;68 is the extended ASCII code for the F10 key; 13 decimal is a carriage return.

- The following example sets the prompt to display the current directory on the top of the screen and the current drive on the current line:

```
prompt $e[s$e[1;30f$e[K$p$e[u$n$g
```

If the current directory is C:\FILES, and the current drive is C, this example would display:

```
C:\FILES
```

```
C>
```

- The following DOS-compatible assembly language program reassigns the F10 key to a DIR B: command followed by a carriage return:

```
TITLE SET ANSI.ASM - SET F10 TO STRING FOR ANSI.SYS

CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME CS:CSEG,DS:CSEG
        ORG     100H
ENTPT:  JMP     SHORT START
STRING  DB      27,'[0;68;"DIR B: ";13P' ;Redefine F10 key
STRSIZ  EQU     $-STRING                ;Length of above message
HANDLE  EQU     1                       ;Pre-defined file. Handle for standard output

START   PROC    NEAR
        MOV     BX,HANDLE                ;Standard output device
        MOV     CX,STRSIZ                ;Get size of text to be sent
        MOV     DX,OFFSET STRING        ;Pass offset of string
        ;To be sent
        MOV     AH,40H                  ;Function="write to device"
        INT     21H                    ;Call DOS
        RET                                ;Return to DOS
START   ENDP

CSEG    ENDS
        END     ENTPT
```

Part 4. Reference Material

Chapter 15. Physical Device Driver Strategy Commands

The physical device driver strategy routine is called with ES:BX pointing to the request packet.

Request Packets

The operating system does not guarantee that the order of API requests issued by multiple threads will be preserved in the order that the corresponding request packets arrive at the physical device driver. Multiple application threads, or threads created due to DosReadAsync and DosWriteAsync, can get blocked in the operating system. This allows a physical device driver request packet (for an API request by a subsequent thread that does not get blocked) to arrive out of order. If a device driver supports multiple outstanding requests, it is responsible for providing a synchronization mechanism between itself, and application processes. Also, request packet ordering must be preserved.

A request packet consists of two parts, the request header and the command-specific data field. The structure of the request packet is detailed below:

Field	Length
Length of Request Packet	BYTE
Block Device Unit Code	BYTE
Command Code	BYTE
Request Packet Status	WORD
Reserved	DWORD
Queue Linkage	DWORD
Command-Specific Data	BYTES

Length of Request Packet: Set to the total length, in bytes, of the request packet (the length of the request header, plus the length of the command-specific data).

Block Device Unit Code: Identifies the unit for which the request is intended. This field has no meaning for character devices.

Command Code: Indicates the requested device driver function. The physical device driver command codes are summarized in "Summary of Strategy Commands" on page 15-4.

Request Packet Status: Defined only for OPEN and CLOSE request packets on entry to the strategy routine. For all other request packets, the Status field is undefined on entry. For an OPEN request packet, bit 3 (08H) of the Status field is set if the packet was generated from a DosMonOpen; otherwise, it is a DosOpen.

For a CLOSE request packet, bit 3 (08H) of the Status field is set, if the packet was generated by a DosMonClose, or a DosClose of a handle that was generated by a DosMonOpen. In this way, monitor handles generated and left open when a process exits are closed properly. Otherwise, it is a DosClose on a non-monitor handle.

PDD strategy commands

On exit from the strategy routine, the Status field describes the resulting state of the request as shown in Table 15-1.

15	14	13-10	9	8	7-0
E R R O R	D E V E R R O R	RESERVED	B U S Y	D O N E	ERROR CODE (bit 15 on)

Table 15-1. Request Packet Status Field

Bit 15: The Error bit. If this bit is set, the low 8 bits of the status WORD (7-0) indicate the error code, which is processed by the operating system in one of the following ways:

- If the IOCTL category is *User Defined*, (refer to the Category Code under Generic IOCTL Commands), FF00H is ORed with the byte-wide error code.
- If not *User Defined* and bit 14 (device driver defined error code) is set, FE00H is ORed with the byte-wide error code.
- Otherwise, the error code must be one of those shown in the table below, and is mapped into one of the standard OS/2 API return codes.

Bit 14: A device driver-defined error, if set in conjunction with bit 15.

Bits 13-10: Reserved.

Bit 9: The *busy* bit. It is only set by Status, and Removable Media. See "6H, AH / INPUT OR OUTPUT STATUS" on page 15-13, and "FH / REMOVABLE MEDIA" on page 15-16 for more information.

Bit 8: The *done* bit. If set, it means the operation is complete. The physical device driver sets the done bit to 1, when exiting.

Bits 7-0: The low 8 bits of the status WORD. If bit 15 is set, bits 7-0 contain the error code. The error codes and errors are shown in Table 15-2.

Table 15-2 (Page 1 of 2). Status Field Error Codes	
Error Codes	Description
00H	Write Protect Violation
01H	Unknown Unit
02H	Device Not Ready
03H	Unknown Command
04H	CRC Error
05H	Bad Drive Request Structure Length
06H	Seek Error
07H	Unknown Media
08H	Sector Not Found
09H	Printer Out of Paper

<i>Table 15-2 (Page 2 of 2). Status Field Error Codes</i>	
Error Codes	Description
0AH	Write Fault
0BH	Read Fault
0CH	General Failure
0DH	Change Disk (logical switch)
0EH	Reserved
0FH	Reserved
10H	Uncertain Media
11H	Character I/O Call Interrupted
12H	Monitors Not Supported
13H	Invalid Parameter
14H	Device Already in Use
15H	Initialization Failed (non-critical)

Uncertain Media (10H): Returned when the state of the media in the drive is uncertain. This response should *not* be returned to the INIT command. For fixed disks, the physical device driver must begin in a media-uncertain state in order to have the media correctly labelled. In general, the following guidelines can be used to determine when to respond with uncertain media:

- When a drive-not-ready condition is detected. In this case, return uncertain media to all subsequent commands, until a reset media command is received.
- When accessing removable media without change-line support, and a time delay of two or more seconds has occurred.
- When the state of the change-line indicates that the media might have changed.

Character I/O Call Interrupted (11H): Returned when the thread performing the I/O was interrupted out of a DevHlp Block, before completing the requested operation.

Monitors Not Supported (12H): Returned for monitor commands (monitor open/close, register IOCtl), if monitors are not supported by the physical device driver.

Invalid Parameter (13H): Returned when one or more fields of the request packet contain invalid values.

Queue Linkage: Provided to maintain a linked list of request packets. The device driver can use the request queue management DevHlp services, or it can use its own queue management.

Command-Specific Data: The parameters required for the physical device driver command. The commands and actual formats of the corresponding request packets are discussed in the following sections.

Summary of Strategy Commands

The following table lists the physical device driver strategy commands:

Code	Function	Block	Char
0H	INIT	X	X
1H	MEDIA CHECK	X	
2H	BUILD BPB	X	
3H	Reserved		
4H	READ (input)	X	X
5H	NONDESTRUCTIVE READ NO WAIT		X
6H	INPUT STATUS		X
7H	INPUT FLUSH		X
8H	WRITE (output)	X	X
9H	WRITE WITH VERIFY	X	X
AH	OUTPUT STATUS		X
BH	OUTPUT FLUSH		X
CH	Reserved		
DH	OPEN DEVICE	X	X
EH	CLOSE DEVICE	X	X
FH	REMOVABLE MEDIA	X	
10H	GENERIC IOCTL	X	X
11H	RESET MEDIA	X	
12H	GET LOGICAL DRIVE MAP	X	
13H	SET LOGICAL DRIVE MAP	X	
14H	DEINSTALL		X
15H	Reserved		
16H	PARTITIONABLE FIXED DISKS	X	
17H	GET FIXED DISK/LOGICAL UNIT MAP	X	
18H	Reserved		
19H	Reserved		
1AH	Reserved		
1BH	Reserved		
1CH	SHUTDOWN	X	X
1DH	GET DRIVER CAPABILITIES	X	

Note: All DWORD pointers are stored with offset first, then segment.

0H / INIT

This command initializes the physical device driver.

Request Block Format

Field	Length
Request Header	13 BYTES
Data_1	BYTE
Pointer_1	DWORD
Pointer_2	DWORD
Data_2	BYTE

Remarks: On entry, the request block contains the following fields as inputs to the physical device driver:

Pointer_1 Points to the DevHlp entry point
 Pointer_2 Points to the INIT arguments
 Data_2 Drive number for the first block device unit.

The arguments for installable device drivers, from the DEVICE = line in the CONFIG.SYS file, allow the physical device driver to use configurable parameters to initialize itself and its device.

At initialization time, the physical device driver runs as a thread under a protect-mode process at application level with I/O privilege. The device driver can issue certain OS/2 dynalink function calls at this time. Refer to "Physical Device Driver Initialization" on page 4-2 for more details. On completion of initialization, the physical device driver must set fields in the request packet as described:

Fields	Output Information for INIT Success
Data_1	Number of logical block devices or units (block devices only)
Pointer_1	WORD offset to end of code segment, and WORD offset to end of data segment
Pointer_2	Points to the BIOS Parameter Block (BPB) array for the logical block devices or units (block devices only)
Status	Set the status WORD in the request header to 0100H.

A block device driver must return in *Data_1*, the number of logical devices or units that are available. The kernel's file system layer assigns sequential drive letters to these units. A character device driver then sets *Data_1* to 0.

Both block device drivers and character device drivers must set *Pointer_1* with the offsets of the code and data segments. This allows a physical device driver to release code and data needed only by the device driver's initialization routine. First, the initialization code and data must be located at the end of the appropriate segments. Then, as the final step in initialization, the physical device driver sets the offsets to the end of the code segment, and the end of the data segment. This also permits a physical device driver to load with a maximum-sized data segment (64 KB), and let it release the amount that it does not need.

A block device driver must return an array of BPBs for the logical units that it supports in *Pointer_2*. A character device driver sets *Pointer_2* to 0.

The *Status* field in the request packet header must be set to indicate *no error*, and *done*. If the physical device driver determines that it cannot set up the device and wants to quit, it should return with the error bit in the request packet status field set to 1. The physical device driver can also return the following:

Fields	Output Information for INIT Failure
Data_1	BYTE 00H
Pointer_1	WORD 0000H, and WORD 0000H
Status	810CH

The Status field in the request packet header must be set to indicate the failure of the INIT request with the *general failure* error return code. The Status must also indicate that the request is done.

One of the above techniques must be used to return device initialization failures from the physical device driver to the system initialization process. A character device driver that contains multiple device driver headers can fail initialization on a subset of the headers in its header chain.

The system initialization process remembers the last non-zero size code and data segment offsets returned for the devices in the device driver that completed initialization. These last values are used to resize the physical device driver's code and data segments, after INIT packets have been sent to the physical device driver for each device in the physical device driver header chain.

When a device in the header chain cannot be initialized, the physical device driver can set the code and data segments to 0, and set the error bit in the request packet Status field to indicate initialization failure for that device. The physical device driver will not receive any future request packets for a specific device, if it returns a failure for the INIT request packet for that device. If none of the devices in the device driver header chain pass initialization, the physical device driver does not remain loaded. Because the system initialization process maintains the pass/fail return status for each device header in a physical device driver header chain, the physical device driver should not manipulate the linkages of the headers.

1H / MEDIA CHECK

This command determines the state of the media.

Request Block Format

Field	Length
Request Header	13 BYTES
Media Descriptor	BYTE
Return Code	BYTE
Return Pointer to Previous Volume ID (if supported)	DWORD

Remarks: On entry, the request packet will have the media descriptor field set for the drive identified in the request packet header. The physical device driver must perform the following actions for the MEDIA CHECK request:

- Set the status WORD in the request packet header
- Set the return code:
 - 1 = Media has been changed
 - 0 = Unsure, if media has been changed
 - 1 = Media unchanged.

Examples of DOS media descriptor bytes are shown below:

Disk Type	Sides	Sectors/Track	Media Descriptor
Fixed disk	--	--	F8H
3.5 inch	2	9	F9H
3.5 inch	2	18	F0H
5.25 inch	2	15	F9H
5.25 inch	1	9	FCH
5.25 inch	2	9	FDH
5.25 inch	1	8	FEH
5.25 inch	2	8	FFH
8 inch	1	26	FEH
8 inch	2	26	FDH
8 inch	2	8	FEH

Examples of DOS Media Descriptor Bytes

Note: To determine whether a single-sided, or a double-sided 8-inch diskette (FEH) is in use, attempt to read the second side. If an error occurs, assume the diskette is single-sided.

For 8-inch diskettes:

FEH (IBM 3740 format)

Single-sided, single density, 128 bytes-per-sector, soft sector, 4 sectors per allocation unit, 1 reserved sector, 2 File Allocation Tables (FATs), 68 directory entries.

FDH (IBM 3740 format) Double-sided, single density, 128 bytes-per-sector, soft sectored, 4 sectors per allocation unit, 4 reserved sectors, 2 FATs, 68 directory entries.

FEH Double-sided, double density, 1024 bytes-per-sector, soft sectored, 1 sector per allocation unit, 1 reserved sector, 2 FATs, 192 directory entries.

Note: Application programmers are encouraged to use the generic IOCTL, “Function 63H – Query Device Parameters” on page 18-159, and reference the BIOS Parameter Block (BPB), to determine the type of media.

2H / BUILD BPB

This command is requested when the media has changed or when the media type is uncertain.

Request Block Format

Field	Length
Request Header	13 BYTES
Media Descriptor	BYTE
Transfer Address	DWORD
Pointer to BPB Table	DWORD
Drive Number	BYTE

Remarks: On entry, the request packet has the media descriptor set for the drive identified in the request packet header. If the block device driver attribute field has bit 13 set, the transfer address is a virtual address to a buffer containing the boot sector media. Otherwise the buffer contains the first sector of the File Allocation Table (FAT).

The physical device driver must perform the following actions:

- Set the pointer to the BPB table
- Update the media descriptor
- Set the status WORD in the request header.

The physical device driver must determine the media type in the drive in order to return the pointer to the BPB table. Previously, the FAT ID byte determined the structure and layout of the media. Because the FAT ID byte has only eight possible values (F8 through FF), it is clear that, as new media types are invented, the available values will soon be exhausted. With the varying media layouts, the operating system should be aware of the location of the FATs and directories, before it reads them.

The physical device driver reads the boot sector from the specified buffer. If the boot sector is for OS/2 or DOS 2.0 (or higher), the device driver returns the BPB from the boot sector. If the boot sector is for DOS 1.00 or 1.10, the physical device driver reads the first sector of the FAT into the specified buffer. The FAT ID is examined and the corresponding BPB is returned. Only two formats are possible for diskettes formatted by a DOS 1.00 or 1.10 system: 5 1/4-inch single-sided (FEH), and 5 1/4-inch double-sided (FFH).

Boot Sector Format: The information relating to the BPB for a particular media is kept in the boot sector for the media. See Table 15-3 on page 15-10.

Field	Length
Short JUMP (EBH) Followed by a NOP (90H)	2 BYTES
OEM Name and Version	8 BYTES
Bytes per Sector	WORD
Sectors per Allocation Unit (must be a power of 2)	BYTE
Reserved Sectors (starting at logical Sector 0)	WORD
Number of FATs	BYTE
Number of Root Directory Entries (maximum allowed)	WORD
Number of Sectors in Logical Image (total sectors in media, including boot sector and directories)	WORD
Media Descriptor	BYTE
Number of Sectors Occupied by a Single FAT	WORD
Sectors per Track	WORD
Number of Heads	WORD
Number of Hidden Sectors	WORD

Table 15-3. Boot Sector Format

The last three WORDs above help the physical device driver understand the media. The number of heads is useful for supporting different multiple head drives that have the same storage capacity, but a different number of surfaces. The number of hidden sectors is useful for supporting drive partitioning schemes.

For drivers that support volume identification and disk change, this command should cause a new volume identification to be read off the disk. This command also indicates that the disk was properly changed.

4H, 8H, 9H / READ/WRITE/WRITE WITH VERIFY

This command reads from or writes to a device (read from, 4H, write to, 8H, and write with verify, 9H).

Request Block Format

Field	Length
Request Header	13 BYTES
Media Descriptor	BYTE
Transfer Address	DWORD
Byte/Sector Count	WORD
Starting Sector Number for Block Device	DWORD
System File Number	WORD

Remarks: On entry, the request packet has the media descriptor set for the drive identified in the request packet header. The transfer address is a 32-bit physical address of the buffer for the data. The byte/sector count is set to the number of bytes to transfer (for character device drivers), or the number of sectors to transfer (for block device drivers). The starting sector number is set for the block device drivers. The System File Number is a unique number associated with an Open request.

The physical device driver must perform the following actions:

- Perform the requested function
- Set the actual number of sectors or bytes transferred
- Set the status WORD in the request header.

The DWORD transfer address in the request packet is a locked 32-bit physical address. The physical device driver can pass it to the DevHlp function, PhysToVirt, to obtain a segment-swapping address for the current mode. The physical device driver does not need to unlock the address when the request is completed.

Note: The IOCTLs, READ and WRITE, are not supported by OS/2 physical device drivers.

5H / NONDESTRUCTIVE READ NO WAIT

This command reads a character from the buffer but does not remove it.

Request Block Format

Field	Length
Request Header	13 BYTES
Returned Character	BYTE

Remarks: The physical device driver must perform the following actions:

- Return a byte from the device
- Set the status WORD in the request header.

For input on character devices with a buffer, the physical device driver returns from this function with the busy bit set to 0, along with a copy of the first character in the buffer. The busy bit is set to 1, to indicate no characters in the buffer. This function allows the operating system to look ahead one input character, without blocking in the physical device driver.

6H, AH / INPUT OR OUTPUT STATUS

This command determines I/O status on character devices (input status, 6H, output status, AH).

Request Block Format

Field	Length
Request Header	13 BYTES

Remarks: The physical device driver must perform the following actions:

- Perform the requested function
- Set the busy bit
- Set the status WORD in the request header.

For output on character devices, if the busy bit is returned set to 1, a subsequent Write request to the physical device driver has to wait for the completion of a currently active request. If the busy bit is set to 0, there is no current request. Therefore, a Write request would start immediately.

For input on character devices with a buffer, if the busy bit is returned set to 1, there are no characters currently buffered in the physical device driver. If the busy bit is set to 0, there is at least one character in the physical device driver buffer. The effect of busy bit=0 is that a read of one character does not need blocking. Devices that do not have an input buffer in the physical device driver should always return busy=0.

7H, BH / FLUSH INPUT OR OUTPUT

This command flushes or terminates all pending requests (input flush, 7H, output flush, BH).

Request Block Format

Field	Length
Request Header	13 BYTES

Remarks: The physical device driver must perform the following actions:

- Perform the requested function
- Set the status WORD in the request header.

This command tells the physical device driver to flush (terminate) all known pending requests. Its primary use is to flush the input (or output) queue on character devices.

DH, EH / OPEN OR CLOSE DEVICE

This command opens or closes the device (open device, DH, close device, EH).

Request Block Format

Field	Length
Request Header	13 BYTES
System File Number	WORD

Remarks: The System File Number is a unique number associated with an open request. The physical device driver must perform the following actions:

- Perform the requested function
- Set the status WORD in the request header.

Character device drivers can use Open/Close requests to correlate using their devices with application activity. For instance, the physical device driver can use the OPEN as an indicator to send an initialization string to its device. The physical device driver can then increase a reference count for every OPEN and decrease the reference count for every CLOSE. When the count goes to 0, the physical device driver can flush its buffers.

For example, to ensure that a printer is in a known state at the start of an I/O stream, this command could be used to set the font and page size. Similarly, the CLOSE call can be used to send a post-string (like a form feed) at the end of an I/O stream. Using an IOCTL to set these pre-strings and post-strings provides a flexible mechanism of serial I/O device stream control.

FH / REMOVABLE MEDIA

This command checks for removable media.

Request Block Format

Field	Length
Request Header	13 BYTES

Remarks: The physical device driver must perform the following actions:

1. Set the busy bit of the status WORD to:
 - 1 – If the media is non-removable
 - 0 – If the media is removable.
2. Set the status WORD in the request header.

The physical device driver receives this request packet, when an application issues an IOCTL function call, to determine whether it is dealing with a removable or non-removable media drive. For example, removable or non-removable drives can print different versions of some prompts.

10H / GENERIC IOCTL

This command sends I/O control commands to a device.

Request Block Format (DosDevIOctl)

Field	Length
Request Header	13 BYTES
Function Category	BYTE
Function Code	BYTE
Parameter Buffer Address	DWORD
Data Buffer Address	DWORD
System File Number	WORD

Request Block Format (DosDevIOctl2)

Field	Length
Request Header	13 BYTES
Function Category	BYTE
Function Code	BYTE
Parameter Buffer Address	DWORD
Data Buffer Address	DWORD
System File Number	WORD
Parameter Buffer Length	WORD
Data Buffer Length	WORD

Remarks: On entry, the request packet has the IOCTL category code and function code set. Parameter Buffer Address and Data Buffer Address are set as virtual addresses. Notice that some IOCTL functions do not require data or parameters to be passed. For these IOCTLs, the parameter and data buffer addresses can contain zeros. The System File Number is a unique number associated with an Open request.

If the physical device driver indicates in the function level of the Device Attribute field of its device header that it supports DosDevIOctl2, the generic IOCTL request packets passed to the physical device driver will have two additional WORDs containing the lengths of the Parameter Buffer and Data Buffer, respectively. If the physical device driver indicates through the function level that it supports DosDevIOctl2, but the application issues DosDevIOctl, the Parameter Buffer and Data Buffer Length fields will be set to 0. The physical device driver must perform the following actions:

- Perform the requested function
- Set the status WORD in the request header.

The physical device driver is responsible for locking the parameter and data buffer segments, and converting the pointers to 32-bit physical addresses, if necessary. Refer to the *OS/2 2.0 Control Program Programming Reference* and the *OS/2 2.0 Programming Guide* for more detailed information on the generic IOCTL interface for applications (DosDevIOctl). Refer to Chapter 18, "Generic IOCTL Commands" on page 18-1 for a detailed description of the generic IOCTL interface for physical device drivers.

11H / RESET MEDIA

This command resets the Uncertain Media error condition and allows the operating system to identify the media.

Request Block Format

Field	Length
Request Header	13 BYTES

Remarks: On entry, the unit code identifies the drive number to be reset. The physical device driver must perform the following actions:

- Set the status WORD in the request header
- Reset the error condition for the drive.

Before this command, the physical device driver had returned the error, *Uncertain Media*, for the drive. This action informs the driver that it no longer needs to return the error for the drive.

12H, 13H / GET OR SET LOGICAL DRIVE MAP

This commands gets or sets the logical drive, which is currently mapped onto a particular unit (get logical drive mapping, 12H, and set logical drive mapping, 13H).

Request Block Format

Field	Length
Request Header	13 BYTES

Remarks: On entry, the unit code contains the unit number of the drive for which this operation is to be performed. The physical device driver must perform the following actions:

- For GET, it must return the logical drive that is mapped onto the physical drive, indicated by the unit number in the request header.
- For SET, it must map the logical drive represented by the unit number onto the physical drive that has the mapping of logical drives.
- The logical drive is returned in the unit code field. This field is set to 0, if there is only one logical drive mapped onto the physical drive.
- Set the status WORD in the request header.

14H / DEINSTALL

This command terminates the character device driver.

Request Block Format

Field	Length
Request Header	13 BYTES

Remarks: When a physical device driver is loaded, the attribute field, and name in its header, are used to determine if the new device driver is attempting to replace a driver already installed. If so, the previously installed device driver is requested by the operating system to DEINSTALL the indicated device. If the installed device driver refuses the DEINSTALL command, the new device driver is not allowed to initialize. If the installed device driver performs the DEINSTALL, the new device driver is initialized.

If the character device driver honors the DEINSTALL request, it must perform the following actions:

- Release any allocated physical memory.
- Unset any hardware vectors that it had claimed.
- Perform any other cleanup.
- Clear the error bit in the Status field to indicate a successful DEINSTALL.

If the character device driver determines that it cannot terminate, it should set the error bit in the Status field, and set the error code to 03H, UNKNOWN COMMAND.

DEINSTALL Considerations

- **Hardware Interrupts.** In honoring a DEINSTALL command, a device driver must remove its claim on the interrupt level. The DevHlp, UnSetIRQ, provides this service.

If the physical device driver's device is ill-behaved (that is, it cannot be told to stop generating interrupts, or be quiesced), the physical device driver must not remove its interrupt handler. In this case, the physical device driver must refuse the DEINSTALL request.

Note: Because of the general interrupt sharing capabilities in a level-sensitive interrupt environment, device drivers should not assume that the DevHlp SetIRQ service can be used to determine whether a given device is being used by another device driver. Instead, the DEINSTALL convention should be used on the logical device name that another driver might be using to access the same device.

16H / PARTITIONABLE FIXED DISKS

This command is used by the system to ask the physical device driver how many physical-partitionable fixed disks the physical device driver supports.

Request Block Format

Field	Length
Request Header	13 BYTES
Count	BYTE
Reserved	WORD
Reserved	WORD

Remarks: This is done to allow the Category 9 generic IOCTLs to be routed appropriately to the correct device driver. This command is not tied to a particular unit that the physical device driver owns, but is directed to the driver as a general query of its device support.

The physical device driver must perform the following actions:

- Set the count as discussed above (1-based)
- Set the status WORD in the request header.

17H / GET FIXED DISK/LOGICAL UNIT MAP

This command is used by the system to determine which logical units supported by the physical device driver exist on the physical partitionable fixed disk, N.

Request Block Format

Field	Length
Request Header	13 BYTES
Units-Supported Bit Mask	4 BYTES
Reserved	WORD
Reserved	WORD

Remarks: On entry the request packet header unit field identifies a physical disk number (based on 0) instead of a logical unit number. The device driver returns a bitmap of which logical units exist on the physical drive. The physical drive relates to the partitionable fixed disks reported to the system by way of the PARTITIONABLE FIXED DISKS command. It is possible that no logical units exist on a given physical disk because it has not yet been initialized.

The physical device driver must perform the following actions:

- Set the 4-byte bit mask to indicate which logical units that it owns, exist on the physical partitionable fixed disk for which the information is being requested.
- Set the status WORD in the request packet header.

The bit mask is set up as follows:

- 0 The logical unit does not exist
- 1 The logical unit exists.

The first logical unit that the physical device driver supports is the low-order bit of the first byte. The bits are used from right-to-left starting at the low-order bit of each following byte. It is possible that all the bits will be 0. As an example, a block device driver supports five units spread over the two diskette drives and one partitionable fixed disk in a system. Unit 0 and unit 1 map to the diskette drives. Unit 2, 3, and 4 map to the fixed disk. For the device command, this physical device driver sets the 4-byte bit map to:

'0000 0000 0000 0000 0000 0000 0001 1100' binary
 or
 '00 00 00 1C' hex

1CH / SHUTDOWN

This command writes buffered data out to the device (flushes the buffer).

Request Block Format

Field	Length
Request Header	13 BYTES
Function Code	BYTE
Reserved	DWORD

Remarks: Each device is called twice. On entry of the START SHUTDOWN call, the function code is set to 0. On entry of the END SHUTDOWN call, the function code is set to 1. The reserved field is set to 0.

The physical device driver must perform the following actions:

- Commit buffered data to device
- Set the status WORD in the request header.

A device driver is called with the system SHUTDOWN request packet, only if it is a level 2 device driver, or if it is a level 3 device driver and has turned on the SHUTDOWN support flag of the Capabilities field found in the physical device driver header.

1DH / GET DRIVER CAPABILITIES

This command returns the functional capabilities of the physical device driver for device drivers supporting the Extended Device Driver interface.

Request Block Format

Field	Length
Request Header	13 BYTES
Reserved. Must be 0.	3 BYTES
DD_CapStruc	DWORD
DD_VolCharStruct	DWORD

Remarks: See “Identifying Extended Device Drivers and Capabilities” on page 16-3 for detailed information regarding this strategy command.

Chapter 16. Extended Device Driver Interface

The Extended Device Driver interface supported in OS/2 2.0, is specifically targeted for service of fixed disk devices. This interface can:

- Support submission of multiple asynchronous requests
- Allow physically discontinuous data transfer areas
- Support a new class of disk devices with advanced bus-mastering and intelligent transfer capabilities
- Allow physical device drivers to optimally service large numbers of requests in a heavily loaded environment
- Provide a mechanism and supporting semantics to support prioritization of request service.

The Extended Device Driver interface employs a Request List of prioritized commands, which the driver can reorder to optimize disk access, subject to prioritization requirements. The requests can also be grouped by the kernel for notification callout from the device driver. In addition, READ and WRITE operations use scatter/gather descriptors, which allow for data transfer to and from discontinuous data buffers. The interface is used asynchronously, removing the need for blocking in the physical device driver, or the physical device driver manager when the I/O request itself is asynchronous.

While the asynchronous, multi-request aspects of the interface contribute greatly to overall system performance on existing hardware, scatter and gather is specifically targeted for new classes of disk device hardware. This new class of devices supports transfer from disk to physically discontinuous memory space much more efficiently than alternative implementations.

The importance of this relative efficiency is amplified by the introduction of paging to the OS/2 operating system, where linearly contiguous memory normally maps to lists of discontinuous physical addresses. In addition, this interface is designed and optimized for server environments, such as LAN Server 2.0, where file service paths are Ring 0 only, and can execute at task or interrupt time. In such an environment, it must be possible to queue requests to the disk driver for service in the context of a network interrupt.

Extended physical Disk device drivers refer to a superset of the standard OS/2 1.x physical Disk device drivers. The term *standard request* is used to refer to the old style request packets format. The term *extended request* is used to refer to the new request packet format.

Disk Device Driver Architecture

Driver architecture centers around the need to provide fast, efficient services to a file system in a paged environment. In addition, consideration for the needs of a network file server has influenced aspects of the design, however, the requirements are identical in a local-only or workstation environment (that is, a direct, asynchronous, zero-overhead interface between the file system and the supporting physical Disk device drivers).

In the OS/2 operating system, a physical Disk device driver receives requests for service through a strategy routine at task time in the context of the requestor (an OS/2 thread). The thread of control is also obtained, in the context of interrupts generated by the disk controller, at the physical device driver interrupt routine. In the extended architecture, a second strategy routine is introduced, which can be called directly by File System Drivers (FSDs) at task time or in the context of an arbitrary interrupt. This yields a set of new requirements.

Standard OS/2 Strategy Routine

The standard OS/2 strategy routine is essentially unchanged in this architecture. Underlying queueing mechanisms used by the driver in the strategy routine are modified to support an environment where there are normally two kinds of requests in the queue, standard and extended.

Extended Strategy Routine

The extended strategy routine entry point can be called at interrupt or task time. At interrupt time, the driver can work only with physical addresses or global virtual addresses; therefore, only physical and global virtual addresses, which reference structures that are physically contiguous in memory, are used in this interface. Physical addresses are used for data transfer areas. Global virtual addresses are used for request packets, control structures, and entry points.

Much of the performance gain realized in this interface is dependent on the asynchronous processing of requests. The performance gain realized is degraded considerably, if the driver blocks in the strategy routine, forcing the request to be synchronous. Therefore, while it is not required that the driver never block, it is very strongly recommended, if performance is of interest in the system for which the driver is targeted. Additionally, if the driver has set the *does-not-block* bit in the DD_DriverCaps field returned from the GET DRIVER CAPABILITIES command, then the driver absolutely must not block in any code path accessible by the extended strategy routine entry point. Furthermore, it must not call any DevHlp routine that could block, effectively limiting the DevHlps that can be called to only those permissible at interrupt time. Among the DevHlps that could block are Lock and Unlock, so all buffers passed through the extended entry points are guaranteed locked.

The extended strategy routine is only used to pass requests in Request List form (see "Request Lists and Request Control" on page 16-6). The physical device driver queues the requests passed, service the controller as necessary, set status fields in the requests, and return.

Sorting and Priority

Requests should be sorted into internal queues based on physical disk location, and I/O priority to optimize request servicing. Lower priority requests should be satisfied, only where they do not significantly slow service of higher priority requests; for example, when a lower priority request refers to a sector in the same cylinder as a high priority request. Since the format of requests in Request Lists is different from the format of standard OS/2 requests, the queue management DevHlp routines cannot be used.

The queueing strategy adopted by the driver is highly dependent on the devices it services. In general, efficiency and request priority are the primary concerns in determining a queueing strategy. Lower priority requests should never slow service of higher priority requests. Lower priority requests can be serviced in the context of servicing higher priority requests, so long as no time-costly operations are necessary to service the lower priority requests. In addition, where contention for resources such as auxiliary, driver-allocated buffers or space on a controller buffer is an issue, higher priority requests should be given preference.

Request Management

The physical device driver needs to manage both requests submitted to the extended strategy routines with the Request List format, and requests submitted to the standard strategy routine with the standard request packet format. Notice that the CommandCode field in the OS/2 standard request packet coincides with the CommandPrefix byte in the extended request packet, which is guaranteed to be the ExecuteChain prefix. This allows the driver to manage both requests on the same queue.

Removable Media

The physical device driver should support requests targeted for removable media in the same way it supports requests for non-removable media.

Devices Not Capable of Scatter/Gather

Although this interface is clearly targeted for a disk device controller capable of scatter/gather, even devices that are not capable of scatter/gather still realize overall system performance gains as a result of supporting multi-request, asynchronous I/O, and interrupt-time execution. How the driver handles emulation of scatter/gather is up to it. In general, emulating scatter/gather to programmed I/O devices introduces no significant overhead.

However, if DMA devices, which do not support scatter/gather, attempt to emulate scatter/gather by mapping each scatter/gather descriptor in a single request to one DMA operation, performance will be poor. The driver is better off staging transfers through a pair of contiguous buffers. One buffer can be serviced by the controller, while the other is being set up for subsequent operations. While there is overhead incurred in block-copying data through the staging buffer, this is no worse than the overhead the file system would incur in contiguous cache blocks before submission to a standard OS/2 device driver.

Identifying Extended Device Drivers and Capabilities

If the physical device driver is an extended device driver, it recognizes the new GET DRIVER CAPABILITIES command. This command returns a characteristics bit field, which describes functional capabilities of the device driver, and an entry point vector, which includes the extended strategy routine and several device driver control routines. If the device driver does not recognize the command, or does not respond appropriately, the kernel and all client FSD assume the physical device driver supports only standard OS/2 requests, and restricts its behavior appropriately.

GET DRIVER CAPABILITIES Command

This command is structured as a standard OS/2 request packet. Note that the fields prefixed by *DD_* are filled in by the physical device driver.

Field	Length
Request Header	13 BYTES
Reserved. Must be zero.	3 BYTES
DD_CapStruc	DWORD
DD_VolCharStruct	DWORD

DD_CapStruc A 16:16 virtual pointer to the Driver Capabilities Structure. This pointer is filled in by the physical device driver. See "Driver Capabilities Structure (DCS)" on page 16-4, for the format of this structure.

DD_VolCharStruct A 16:16 virtual pointer to the Volume Characteristics Structure (VCS) for this volume. This pointer is filled in by the physical device driver. See "Volume Characteristics Structure (VCS)" on page 16-5 for the format of this structure.

GET DRIVER CAPABILITIES

Driver Capabilities Structure (DCS): The Driver Capabilities Structure (DCS) is maintained by the physical device driver, and is passed by reference to the kernel and client FSDs in the GET DRIVER CAPABILITIES command. The kernel and client FSDs must not modify the structure, as it is shared by among FSDs and the physical device driver. A DCS has the following format:

Field	Length
Reserved. Must be zero.	WORD
DD_VerMajor	BYTE
DD_VerMinor	BYTE
DD_Capabilities	DWORD
DD_Strategy2	DWORD
DD_SetFSDInfo	DWORD
DD_ChgPriority	DWORD
DD_SetRestPos	DWORD
DD_GetBoundary	DWORD

- DD_VerMajor** The major version number of the interface the physical device driver supports, equal to 01H in the first release. Old major versions do not function correctly with a file system using a newer version.
- DD_VerMinor** The minor version number of the interface the physical device driver supports, equal to 01H in the first release. Old minor versions support a strict subset of the functionality found in newer versions.
- DD_Capabilities** A bit field describing the capabilities of the physical device driver:
- Bits 0-2** Reserved. Must be zero.
 - Bit 3** If set, supports disk mirroring.
 - Bit 4** If set, supports disk duplexing.
 - Bit 5** If set, driver does not block in Strategy2. LAN Server and LAN Manager products using the HPFS 386 file system require that bit 5 be set to guarantee that the physical device driver does not block in Strategy2.
 - Bits 6-31** Reserved. Must be zero.
- DD_Strategy2** The 16:16 entry point for the strategy routine that supports multi-request asynchronous I/O.
- DD_SetFSDInfo** The 16:16 entry point for DD_SetFSDInfo. The value returned is 0:0, if the service is not provided by the physical device driver.
- DD_ChgPriority** The 16:16 entry point for DD_ChgPriority. The value returned is 0:0, if the service is not provided by the physical device driver.
- DD_SetRestPos** The 16:16 entry point for DD_SetRestPos. The value returned is 0:0, if the service is not provided by the physical device driver.
- DD_GetBoundary** The 16:16 entry point for DD_GetBoundary. The value returned is 0:0, if the service is not provided by the physical device driver.

Volume Characteristics Structure (VCS): The parameters passed in the Volume Characteristics Structure (VCS) are used by FSDs to optimize disk access, and placement of file system structures on an advisory basis. All values reflect the physical parameters of the logical volume, as if it were a single physical device (that is, whether the media is partitioned or not). This data structure is passed by reference, and is maintained and updated by the physical device driver, as necessary. It is expected that the physical device driver would maintain separate VCSs for each logical volume supported. A VCS has the following format:

Field	Length
VolDescriptor	WORD
AvgSeekTime	WORD
AvgLatency	WORD
TrackMinBlocks	WORD
TrackMaxBlocks	WORD
Head Per Cylinder	WORD
VolCylinderCount	DWORD
VolMedianBlock	DWORD
MaxSGList	WORD

VolDescriptor

A bit field, defined as follows:

- Bit 0** If set, volume resides on removable media
- Bit 1** If set, volume is read-only
- Bit 2** If set, average seek time independent of position (RAM disk)
- Bit 3** If set, outboard cache supported
- Bit 4** If set, scatter/gather supported by adapter
- Bit 5** If set, ReadPrefetch supported
- Bits 6-15** Reserved, set to 0.

AvgSeekTime

The average seek time in milliseconds in servicing this volume. If the seek time is unknown, FFFFH is to be specified. Can be 0 for RAM disks.

AvgLatency

The average rotational latency in milliseconds for the device servicing this volume. If the average latency is unknown, FFFFH is to be specified. Latency can be 0 for RAM disks.

TrackMinBlocks

The number of blocks available on the smallest capacity track; if unknown or not applicable, a value of 1 is specified.

TrackMaxBlocks

The number of blocks available on the largest capacity track; if unknown or not applicable, a value of 1 is specified.

Heads Per Cylinder

The number of heads per cylinder; if unknown or not applicable, a value of 1 is specified.

VolCylinderCount

The number of cylinders in the volume; if unknown or not applicable, the number of allocation blocks (sectors) is used.

VolMedianBlock

The number of the block, which is in the center of the volume with respect to seek time; that is, the block with the smallest average seek time.

MaxSGList

The maximum number of scatter and gather list entries, which can be directly submitted to the adapter servicing this volume with one low-level I/O command. File systems submitting extended commands with scatter and gather lists > MaxSGList entries, must ensure that the cumulative byte count of each MaxSGList entry in the list is a multiple of the sector size. This field is set to 0, if the volume is serviced by an

Request Lists

adapter, which does not directly support scatter/gather lists. See "Scatter/Gather Descriptor" on page 16-12 for details on scatter/gather lists passed in extended requests.

Request Lists and Request Control

In order to support multi-request asynchronous I/O, a new request format has been defined, called Request Lists. This format allows multiple requests to be submitted in one call to the extended strategy routine, as well as grouping of those requests for notification purposes. In general, requests from any and all lists can be reordered, and considered independently by the physical device driver for optimal throughput. Presently, only READ, WRITE, and READ PREFETCH commands have been defined in the new format.

Through Request Control flags, optional restrictions can be set on the requests to force sequential execution, and to allow early termination of request processing, should any of the requests fail.

The notification mechanism allows the kernel and client file systems to receive callout notification, when specific individual requests complete, when the entire request list completes, or both. In addition, notification can take place, when an error condition occurs, when requests complete successfully, or both. Alternatively, no callout notification can be specified, allowing the system to poll for request completion during idle time.

Extended disk driver requests are submitted directly through the extended strategy routine entry point, DD_Strategy2, obtained through the GET DRIVER CAPABILITIES command, and passed to client FSDs through FS_MOUNT. Requests are submitted in request list format, with ES:BX containing a global pointer to the Request List.

Each request in the list is an EXECUTE CHAIN command, containing the EXECUTE CHAIN command prefix at the same offset into the extended request packet, as the Command field in the standard request packet.

Request Lists have the following format:

Field	Length
Request List Header	20 BYTES
Requests	ARRAY

The Request List header has the following format:

Field	Length
ReqListCount	WORD
Reserved	WORD
LstNotifyAddress	DWORD
LstRequestControl	WORD
Block Device Unit	BYTE
LstStatus	BYTE
DDReserved	DWORD
DDReserved	DWORD

ReqListCount The number of requests in the Request List.

Reserved Must be 0.

LstNotifyAddress The 16:16 address of a notification routine to be called (according to the flags in LstRequestControl), when all requests have been completed or terminated, due to error conditions. LstNotifyAddress is not valid if bits 4 and 5 of LstRequestControl are clear. LstNotifyAddress is called with the following parameters:

ES:BX 16:16 Address of Request List header
CF Set, if an unrecoverable error has occurred.

The physical device driver is responsible for saving and restoring any registers that must survive the call.

LstRequestControl A bit field of control flags as follows:

- Bit 0** Reserved.
 - Bit 1** If set, there is only one request in the list. The same mechanism is used to submit one or many requests.
 - Bit 2** If set, requests to be executed in sequence. Indicates that requests in this list must be executed in the order they appear in the Request List. They need not be executed adjacently, requests from other lists can interleave execution of this list.
 - Bit 3** If set, terminate on error. Indicates that, if an unrecoverable error occurs in processing any request in the list, outstanding requests in the list must not be executed. All Status, ErrorCode, and BlocksXferred fields must be updated, however.
 - Bit 4** If set, notify immediately on error only. Indicates that the request list notification routine, LstNotifyAddress, should be called immediately, if an unrecoverable error occurs in servicing any of the requests in the Request List.
 - Bit 5** If set, notify on completion. Indicates that the request list notification routine should be called, when execution of the requests has completed, regardless of error conditions.
- If bit 4 and bit 5 are clear, the request list notification routine address is not valid. If bit 4 or bit 5 is set, the notification routine address is valid.

Bits 6 – 15 Reserved. Must be 0.

Block Device Unit The logical unit number of the volume the requests are directed to. Note this forces all requests in the list to be addressed to the same block device unit.

LstStatus Indicates the overall status for the Request List. This field should be set immediately by the physical device driver at strategy time, and updated as requests complete successfully or unsuccessfully.

The low nibble indicates the completion status of the requests in the list, giving the LstStatus byte the following values:

- X0H** Indicates that no requests have been queued. It is guaranteed that the kernel will set this status on entry to the physical device driver. The physical device driver sets this status on return, only if queueing was not possible due to exhausted driver-internal resources.
- X1H** Indicates that some requests have not yet been sorted into internal queues. That is, queueing is in process, but has not yet been completed.
- X2H** Indicates that all requests in the list have been queued, and are awaiting service.
- X4H** Indicates that all requests in the list have been completed successfully, or unsuccessfully due to error conditions.

Request Lists

X8H Reserved.

The high nibble indicates the error status of the requests in the list, giving the LstStatus byte the following values:

0XH No error.

1XH Indicates that an error has occurred in processing at least one of the requests, but the physical device driver has successfully recovered the error through retries, ECC, disk mirroring, or duplexing.

2XH Indicates that an unrecoverable error has occurred in processing at least one of the requests.

3XH An unrecoverable error has occurred with retry.

4XH Reserved.

8XH Reserved.

Bits in the high nibble can be set in combination with bits in the low nibble to indicate the various error and completion states. LstStatus is guaranteed to be 0, when the request list is submitted to the physical device driver.

DDReserved Reserved for device driver use in tracking request completion. Since the number of requests in the list is variable, this field might be used to point to an auxiliary structure maintained by the device driver.

Each request has a fixed length header, followed by a variable length, command-specific area. The general format of an extended request is:

Field	Length
Request Header	32 BYTES
Command-Specific	BYTE

General Extended Request Format

Extended Commands

The following extended commands, and corresponding command codes, are defined for use in Request Lists:

1EH READ
1FH WRITE
20H WRITE VERIFY
21H READ PREFETCH.

The format of the command-specific portion of the request packet format along with details of each command are described in the sections that follow.

Request Header: Each request in the Request List begins with a fixed length header, which has the following format:

Field	Length
Request Length	WORD
Command Prefix = 1CH	BYTE
Command Code	BYTE
Header Offset	DWORD
Request Control	BYTE
Priority	BYTE
Status	BYTE
Error Code	BYTE
Notify Address	DWORD
Hint Pointer	DWORD
DDReserved	DWORD
DDReserved	DWORD
DDReserved	DWORD

Request Length The offset of the next request. If this is the last request, the value is FFFFH.

Command Prefix At the same offset in this request header as the command code in the standard OS/2 request header. This byte is always set to EXTENDED REQUEST (1CH), to allow the physical device driver to maintain one queue for both standard and extended requests, while distinguishing the two packet types.

Command Code The request command code. Can have any of the values defined in "Extended Commands" on page 16-8.

Header Offset The offset from the beginning of the Request List header to the header of this request. This field, when subtracted from the address of the header of this request, yields the header of the list. This provides fast access to the control information in the header.

Request Control A bit field of control flags as follows:

Bits 0 – 3 Reserved. Must be 0.

Bit 4 If set, notify on error only. Indicates that the individual request notification routine, NotifyAddress, should be called immediately, in the event an unrecoverable error occurs in servicing the request.

Bit 5 If set, notify on completion. Indicates that the individual notification routine, NotifyAddress, should be called, when execution of the request has completed successfully, and possibly with a recoverable error. This bit does not indicate notification in the case of an unrecoverable error.

If bit 4 and bit 5 are clear, the individual request notification routine address is not valid. If bit 4 or bit 5 is set, the notification routine address is valid.

Bits 4 and 5 can both be set to indicate notification in case of error or successful completion.

Bits 6-7 Reserved. Must be 0.

Priority A bit field indicating the priority of the request. The following values are currently defined, and others may be added as needed, without notice:

00H Prefetch Requests.

- 01H** Low priority request to be satisfied in the context of servicing other, higher priority requests or when no other work exists (lazy-write).
- 02H** Read ahead, low priority pager I/O.
- 04H** Background synchronous user I/O.
- 08H** Foreground synchronous user I/O.
- 10H** High priority pager I/O.
- 80H** Urgent request; all requests at this priority should be satisfied in a single sweep of the disk; no stopping allowed at cylinders other than those necessary to satisfy requests in this priority. The kernel uses this priority in cases such as an impending power failure or shutdown.

The kernel or client FSD can request a priority change after the initial submission of the request to the physical device driver by issuing a call to DD_ChgPriority.

Status

A bit field indicating the status of the request. The low nibble indicates the completion status of the request, giving the Status byte the following values:

- X0H** Not yet queued
- X1H** Queued and waiting
- X2H** In process
- X4H** Done
- X8H** Reserved.

The high nibble indicates the error status of the request, giving the Status byte the following values:

- 0XH** No error
- 1XH** A recoverable error has occurred
- 2XH** An unrecoverable error has occurred
- 3XH** An unrecoverable error has occurred
- 4XH** The request was abnormally ended
- 8XH** Reserved.

High and low nibbles can be set in combination by the physical device driver to indicate combinations of error and status conditions. For example, a code of 12H indicates that a recoverable error has occurred, and the request is still in progress. An error condition indicates a valid error code in the ErrorCode field. Status is guaranteed to be 00H, when the request is submitted.

Error Code

Contains a valid error condition, if an error status is indicated in Status. The following error codes are possible if the error is unrecoverable, and are compatible with previous OS/2 error returns:

- 00H** Write-protect violation
- 01H** Unknown unit
- 02H** Device not ready
- 03H** Unknown command
- 04H** CRC error
- 06H** Seek error
- 07H** Unknown media
- 08H** Block not found
- 0AH** Write fault
- 0BH** Read fault
- 0CH** General failure
- 10H** Uncertain media
- 13H** Invalid parameter.

The following error codes are possible, if the error is recoverable:

- 1AH** Verify error on write, succeeded after retry
- 2AH** Write error, write to mirrored or duplexed drive succeeded

- 3AH** Write error on mirrored or duplexed drive; write to primary drive succeeded
1BH Read error, corrected using ECC
2BH Read succeeded after retry
3BH Read error, recovered from mirrored or duplexed drive.

Notify Address The address of a notification routine to be called (according to the flags in RequestControl), when the request has completed successfully or unsuccessfully due to error conditions. NotifyAddress is not valid if bits 4 and 5 of RequestControl are clear. NotifyAddress is called with the following parameters:

ES:BX 16:16 Address of the request header
 CF Set, if an unrecoverable error has occurred.

The physical device driver is responsible for saving and restoring any registers that must survive the call.

Hint Pointer A 16:16 pointer to a request packet in a Request List. This field can be used when the kernel or the client FSD determines that this request might be grouped best with another request it has already submitted to the physical device driver. The request might have already completed, so the physical device driver must validate that the pointer points to a request on its internal queues. This field is FFFF:FFFFH, if it is unused, that is, if a hint is not being passed.

DDReserved Fields reserved for device driver use.

Write/Read/WriteVerify: The format of the request packet for the WRITE, READ, and WRITE VERIFY commands is:

Field	Length
Request Header	32 BYTES
Start Block	DWORD
Block Count	DWORD
BlocksXferred	DWORD
Flags	WORD
SGDescrCount	WORD
Reserved	DWORD
SGDescriptors	ARRAY

Request Header The fixed length request header.

Start Block The starting disk block for the data transfer operation. A disk block is defined as a 512-byte logical disk sector.

Block Count The number of 512-byte blocks to be transferred.

BlocksXferred The number of 512-byte blocks successfully transferred by the driver. This field is updated before the request and request list notification routines are called, and before the Status field is marked as *Done*.

Flags A bit field of command-specific control flags. The following flags have been defined:

- Bit 0** If set, write through. Defeats any *lazy-write* caching performed by the physical device driver. Notice that lazy-write, through battery-backed RAM, is permitted, even if this bit is set.
Bit 1 If set, cache request on outboard controller cache.
Bits 2 – 15 Reserved, set to 0.

SGDescrCount The number of scatter/gather descriptors in the SGDescriptors field.

Request Lists

SGDescriptors An array of scatter/gather descriptors describing the buffers for data transfer specified by the command.

The File System Driver (FSD) guarantees the following to be true:

BlockCount * 512 equals the sum of the BufferSize fields in SGDescriptors.

In addition, buffers are typically DWORD aligned. The physical device driver should be optimized for this case, but should not rely upon it.

Scatter/Gather Descriptor: READ and WRITE operations use an array of scatter/gather descriptors to describe the buffer space to be used in the operation. This enables transfers of contiguous disk blocks into physically discontinuous, byte-aligned memory blocks. Scatter/gather descriptors have the following format:

Field	Length
BufferPtr	DWORD
Buffer Size	DWORD

BufferPtr A 32-bit physical pointer to the buffer

Buffer Size Size of the buffer in bytes.

READ PREFETCH: READ PREFETCH is defined to take advantage of a two-tiered disk caching scheme, where the first tier is the file system, and the second tier is the controller buffer. This command is supported optionally, if the Read Prefetch bit is set in the VolDescriptor bit field of the VCS for the device. If this bit is set, it is assumed that:

- The physical device driver manages a cache located on the controller.
- A Read into controller memory, followed by a Read into system memory, is less expensive (in terms of host CPU utilization) than just a Read into system memory .

If both of these conditions are not met, then the physical device driver does not publish READ PREFETCH capabilities, because it is more efficient for the file system to perform *read-ahead* into its own cache. READ PREFETCH commands are the lowest priority requests submitted to the physical device driver through the extended strategy routine, and are never serviced prior to other Read/Write requests.

The format of the request packet for READ PREFETCH is:

Field	Length
Request Header	32 BYTES
Start Block	DWORD
Block Count	DWORD
BlocksXferred	DWORD
Flags	WORD
Reserved. Must be 0.	WORD

Request Header The fixed length request header.

Start Block The starting disk block for the data prefetch operation. A disk block is defined as a 512-byte logical disk sector.

Block Count The number of 512-byte blocks to be prefetched.

- BlocksXferred** The number of 512-byte blocks successfully prefetched by the physical device driver. This field is updated before the request and request list notification routines are called, and before the Status field is marked as *Done*.
- Flags** A bit field of command-specific control flags. The following flags have been defined:
- Bit 0** If set, hold only until read. The physical device driver retains the data in controller prefetch buffers only until it is read once. This is to prevent redundant caching in the controller.
 - Bits 1-15** Reserved, set to 0.

Request Control Functions

Request control functions are used by the FSD to manage requests after they have been submitted, to obtain advisory information from the physical device driver, and to pass advisory information to the physical device driver. Entry points are exported by the physical device driver through the GET DRIVER CAPABILITIES command. Request control functions can be called at interrupt time, and cannot block. Request control functions need only preserve segment registers.

The following request control functions are defined:

- DD_SetFSDInfo
- DD_ChgPriority
- DD_SetRestPos
- DD_GetBoundary.

DD_SetFSDInfo: This entry point allows the FSD to inform the physical Disk device driver of its FSD_EndOfInt and FSD_AccValidate entry points. The physical Disk device driver allows DD_SetFSDInfo to be called exactly once, and ignores subsequent calls.

ENTRY

ES:BX 16:16 pointer to the FSDInfo structure.

EXIT

CF Set, if call was ignored.

The format of the FSDInfo structure is:

Field	Length
Reserved. Must be 0.	DWORD
FSD_EndOfInt	DWORD
Reserved. Must be 0.	DWORD
FSD_AccValidate	DWORD

FSD_EndOfInt The 16:16 entry point of the FSD's FSD_EndOfInt routine. This field is set to 0 if the FSD does not provide an FSD_EndOfInt routine. The entry point is called by the physical device driver when it has completed interrupt processing and after it has called DevHlp_EOI. FSD_EndOfInt takes no parameters, and leaves all registers intact.

FSD_AccValidate The 16:16 entry point of the FSD's FSD_AccValidate routine. This field is set to 0 if the FSD does not provide an FSD_AccValidate routine. The entry point is called whenever direct I/O is done through a Category 8 or 9 IOCTL to an HPFS volume, or whenever direct I/O is done to the Master Startup (Boot) record through a Category 9 IOCTL.

For Category 9 IOCTLs, the physical device driver must use the Head, Cylinder, and Sector values passed in the IOCTL, to determine whether the I/O request falls within an HPFS volume, since the unit number in the IOCTL represents the entire physical disk, and not the logical volume.

Request Control Functions

The physical device driver should call `FSD_AccValidate` with:

AL Operation Code
00 Non-destructive (READ, VERIFY)
01 Destructive (WRITE, FORMAT TRACK, and so forth)

On return from the FSD:

NC I/O access is allowed.
CF If set, access is denied. The physical device driver should return error code 00H (Write-protection violation) to the caller.

DD_ChgPriority: This entry point allows the FSD to notify the physical device driver of a possible change in the priority of a request. HPFS calls this entry point with:

ENTRY

ES:BX Address of the request
AL New priority for the request;

EXIT

CF Set, if request packet not found on any of the physical device driver's internal queues

This call is used to change the priority of a previously submitted request. The physical device driver performs whatever resorting of internal queues is necessary, and returns.

The FSD guarantees that the pointer passed, references a valid request, that is, a request with allowed values in all fields. There is no guarantee that the priority of the request has actually been changed, or that the request is still on the internal queues of the driver. If the request has been removed from internal queues, has already been incorporated into internal structures in preparation for service, or has already been serviced, the physical device driver can ignore the requested change.

DD_SetRestPos: This entry point advises the physical device driver of a block to seek, when there is no work in the queue. No immediate action is necessary when the call is made. This call is purely advisory, and can be ignored by the driver, if it is not useful or applicable to the hardware it supports.

ENTRY

AX:BX Block to use as resting point, FFFF:FFFFH, if none
CL Logical Unit Number (A:=0)

EXIT

CF Set, if block is out of range.

The physical device driver updates a static variable, specifying where to rest the heads during idle time. When any seek occurs, either as a result of this call, or as the result of I/O requests, the variable is set to FFFFFFFFH. A value of FFFFFFFFH indicates *rest-where-you-end-up*.

This call essentially assumes that there is only one active logical volume serviced by the underlying physical device. *Physical device*, in this context, means mechanical, usually multi-headed, disk arm. If this is not the case, this call should be ignored by the driver.

DD_GetBoundary: This entry point returns the first block number, which is greater than the block number specified in the DWORD passed, and which is past an access time boundary, such as a cylinder. This information can be used by file systems to place file system objects optimally.

ENTRY

AX:BX Reference block number

EXIT

AX:BX Number of first block past access time boundary
CF Set, if block number out of range.

If the physical device driver cannot compute this efficiently, it can precompute this information and retain it internally, or if this is not feasible, it can return (AX:BX) + 1.

Chapter 17. Device Helper (DevHlp) Services

Many of the functions of an OS/2 physical device driver are related to system operations rather than to hardware operations. An interface to operating system services is available to physical device drivers through the *DevHlp Interface*.

The Device Helper (DevHlp) services are listed alphabetically at the end of this chapter with explanations of the purpose, parameters, and calling conventions of each function.

Using DevHlp Services

Access to these system services is obtained during device driver initialization. The request packet for the INIT command contains a pointer to the entry point for the DevHlp interface. This pointer is saved.

Calling the DevHlp Interface Routine

In order to call a Device Helper service, it must be invoked by:

1. Setting up the appropriate registers. Notice that the DevHlp services pass parameters in registers, as opposed to on the stack.
2. Loading a function code into the DL register.
3. Making a FAR CALL to the DevHlp interface routine (whose address was supplied at device driver initialization time), as in:

```
CALL [Device_Help]
```

Register Usage

All registers except the Flags register are preserved across calls to DevHlp services unless specified as containing return parameters.

State of the Interrupt Flag

The physical device driver can assume that the state of the interrupt flag is preserved, and that the DevHlp routine does not enable interrupts unless stated otherwise in the functional description for each routine. The only exceptions apply to functions that allow the physical device driver to relinquish control of the CPU. Therefore, during calls to functions, such as Yield and TCYield, it cannot be assumed that interrupts will remain disabled.

Constant Definitions

In the syntax examples in the following references, the DevHlp functions are called by placing a defined constant into the DL register, as in:

```
MOV DL, DevHlp_ABIOSCall
```

These DevHlp_* constants are defined in the DEVHLP.INC header file.

16:16 Virtual Address Conversion

Some of the new DevHlp services require 32-bit linear (flat) addresses. The DevHlp service, VirtToLin, must be called to convert a 16:16 virtual address to a 32-bit linear address. Refer to the appropriate DevHlp service for the type of addressing mode required.

New DevHlp Services

OS/2 2.0 introduces several new DevHlp services:

New DevHlp Services	Description
AllocateCtxHook	Allocate a context hook
ArmCtxHook	Arm a context hook
Beep	Generate a beep
CloseEventSem	Close a 32-bit shared event semaphore
DynamicAPI	Dynamically adds a Ring 0 system API.
FreeCtxHook	Free a context hook
FreeGDTSelector	Free selector allocated with AllocateGDTSelector
GetDescInfo	Return information on the contents of descriptor
GetDeviceBlock	Get BIOS device block
LinToGDTSelector	Convert a linear address to a virtual address
LinToPageList	Return the physical pages mapped by a linear range
OpenEventSem	Open a 32-bit shared event semaphore
PageListToGDTSelector	Map given physical addresses to selector
PageListToLin	Map given physical address to a linear address
PostEventSem	Post a 32-bit shared event semaphore
PhysToGDTSel	Map a physical address to a GDT selector
RegisterBeep	Register physical device driver's beep service entry point
RegisterPDD	Register a 16:16 physical device driver for PDD/VDD communication
RegisterTmrDD	Return address of Timer value and rollover count
ResetEventSem	Reset a 32-bit shared event semaphore
VirtToLin	Convert a selector:offset to a linear address
VMAIloc	Allocate a block of physical memory
VMFree	Free memory or a mapping
VMGlobalToProcess	Map a global address into process address space
VMLock	Lock a linear address range of memory within a segment
VMProcessToGlobal	Map a process address into global address space
VMSetMem	Commit or decommit physical memory
VMUnlock	Unlock a (linear address) range of memory within a segment

DevHlp Services and Function Codes

All the DevHlp services, listed by function codes, are shown below:

DevHlp Service	Code	Description
SchedClockAddr	0	Get system clock routine
DevDone	1	Device I/O complete
Yield	2	Yield the CPU
TCYield	3	Yield the CPU to time-critical
Block	4	Block thread on event
Run	5	Unblock thread
SemRequest	6	Claim a semaphore
SemClear	7	Release a semaphore
SemHandle	8	Get a semaphore handle
PushReqPacket	9	Add request to list
PullReqPacket	A	Remove request from list
PullParticular	B	Remove a specific request from list
SortReqPacket	C	Insert request in sorted order to list
AllocReqPacket	D	Get a request packet
FreeReqPacket	E	Free request packet
QueueInit	F	Initialize character queue
QueueFlush	10	Clear character queue
QueueWrite	11	Put a character in the queue
QueueRead	12	Get a character from the queue
Lock	13	Lock memory segment
Unlock	14	Unlock memory segment
PhysToVirt	15	Map a physical-to-virtual address
VirtToPhys	16	Map a virtual-to-physical address
PhysToUVirt	17	Map a physical address to an application's address space
AllocPhys	18	Allocate physical memory
FreePhys	19	Free physical memory
SetROMVector	1A	Set software interrupt vector
SetIRQ	1B	Set a hardware interrupt handler
UnSetIRQ	1C	Reset a hardware interrupt handler
SetTimer	1D	Set a timer handler
ResetTimer	1E	Remove a timer handler
MonitorCreate	1F	Create a monitor
Register	20	Install a monitor
DeRegister	21	Remove a monitor
MonWrite	22	Pass data records to monitor
MonFlush	23	Remove all data from data stream

DevHlp Service	Code	Description
GetDOSVar	24	Return pointer to DOS variable
SendEvent	25	Indicate an event
ROMCritSection	26	ROM BIOS critical section
VerifyAccess	27	Verify memory access
Reserved	28	
Reserved	29	
AttachDD	2A	Obtain a device driver's IDC entry point
InternalError	2B	Signal an internal error
Reserved	2C	
AllocGDTSelector	2D	Allocate GDT descriptors
PhysToGDTSelector	2E	Map a physical to virtual address in GDT
RealToProt	2F	Switch from real mode to protect mode
ProtToReal	30	Switch from protect mode to real mode
EOI	31	Issue an End-Of-Interrupt
UnPhysToVirt	32	Mark PhysToVirt complete
TickCount	33	Modify timer
GetLIDEntry	34	Get Logical ID
FreeLIDEntry	35	Release Logical ID
ABIOSCall	36	Invoke ABIOS function
ABIOSCommonEntry	37	Invoke ABIOS common entry point
GetDeviceBlock	38	Get ABIOS device block
RegisterStackUsage	3A	Indicate stack usage
VideoPause	3C	Suspend/resume video active threads
SAVE_MESSAGE	3D	Display message (for base device drivers)
RegisterPDD	50	Register a 16:16 physical device driver for PDD/VDD communication
RegisterBeep	51	Register a physical device driver's Beep service entry point
Beep	52	Generate a beep
FreeGDTSelector	53	Free selector allocated with AllocateGDTSelector
PhysToGDTSel	54	Map a physical address to a GDT selector
VMLock	55	Lock a linear address range of memory within a segment
VMUnlock	56	Unlock a linear address range of memory within a segment
VMAIloc	57	Allocate a block of physical memory
VMFree	58	Free memory or a mapping
VMProcessToGlobal	59	Map a process address into global address space
VMGlobalToProcess	5A	Map a global address into process address space
VirtToLin	5B	Convert a selector:offset to a linear address
LinToGDTSelector	5C	Convert a linear address to a virtual address
GetDescInfo	5D	Get information on the contents of a descriptor
LinToPageList	5E	Get the physical pages mapped by a linear range

DevHlp Service	Code	Description
PageListToLin	5F	Map given physical address to a linear address
PageListToGDTSelector	60	Map given physical addresses to a selector
RegisterTmrDD	61	Get the kernel address of Timer value and rollover count
AllocateCtxHook	63	Allocate a context hook
FreeCtxHook	64	Free a context hook
ArmCtxHook	65	Arm a context hook
VMSetMem	66	Commit or decommit physical memory
OpenEventSem	67	Open a 32-bit shared event semaphore
CloseEventSem	68	Close a 32-bit shared event semaphore
PostEventSem	69	Post a 32-bit shared event semaphore
ResetEventSem	6A	Reset a 32-bit shared event semaphore
DynamicAPI	6C	Dynamically add a Ring 0 system API

DevHlp Services and Device Contexts

As discussed in the section on physical device driver architecture, device driver code can run in one of the following contexts:

- Kernel mode** Context in which the physical device driver strategy routine runs. Also called *task time*.
- Interrupt mode** Context in which the physical device driver hardware interrupt handler runs.
- INIT mode** Context in which the physical device driver strategy routine runs when it is called with the INIT request packet.

Certain restrictions apply, relating to when individual DevHlp services can be used. The table below shows the contexts in which each DevHlp service can be called:

DevHlp Service	Code	Kernel	Interrupt	INIT
SchedClockAddr	0	X		X
DevDone	1	X	X	
Yield	2	X		
TCYield	3	X		
Block	4	X		
Run	5	X	X	
SemRequest	6	X		
SemClear	7	X	X	
SemHandle	8	X	X	
PushReqPacket	9	X		
PullReqPacket	A	X	X	
PullParticular	B	X	X	
SortReqPacket	C	X		
AllocReqPacket	D	X		
FreeReqPacket	E	X		
QueueInit	F	X	X	X
QueueFlush	10	X	X	
QueueWrite	11	X	X	
QueueRead	12	X	X	
Lock	13	X		X
Unlock	14	X		X
PhysToVirt	15	X	X	X
VirtToPhys	16	X		X
PhysToUVirt	17	X		X
AllocPhys	18	X		X
FreePhys	19	X		X
SetROMVector	1A	X		X
SetIRQ	1B	X		X
UnSetIRQ	1C	X	X	X

DevHlp Service	Code	Kernel	Interrupt	INIT
SetTimer	1D	X		X
ResetTimer	1E	X	X	X
MonitorCreate	1F	X		X
Register	20	X		
DeRegister	21	X		
MonWrite	22	X	X	
MonFlush	23	X		
GetDOSVar	24	X		X
SendEvent	25	X	X	
ROMCritSection	26			
VerifyAccess	27	X		
AttachDD	2A	X		X
InternalError	2B	X	X	X
AllocGDTSelector	2D			X
PhysToGDTSelector	2E	X	X	X
RealToProt	2F	X	X	
ProtToReal	30	X	X	
EOI	31		X	X
UnPhysToVirt	32	X	X	X
TickCount	33	X	X	X
GetLIDEntry	34	X		X
FreeLIDEntry	35	X		X
ABIOSCall	36	X	X	X
ABIOSCommonEntry	37	X	X	X
RegisterStackUsage	3A			X
VideoPause	3C	X	X	X
SAVE_MESSAGE	3D			X
GetDeviceBlock	38			X
RegisterPDD	50	X		X
RegisterBeep	51	X		X
Beep	52	X	X	X
FreeGDTSelector	53	X		X
PhysToGDTSel	54	X	X	X
VMLock	55	X		X
VMUnlock	56	X		X
VMAIloc	57	X		X
VMFree	58	X		X
VMProcessToGlobal	59	X		
VMGlobalToProcess	5A	X		

device helper services

DevHip Service	Code	Kernel	Interrupt	INIT
VirtToLin	5B	X	X	X
LinToGDTSelector	5C	X	X	X
GetDescInfo	5D	X	Note 1	X
LinToPageList	5E	X	X	X
PageListToLin	5F	X	X	X
PageListToGDTSelector	60	X	X	X
RegisterTmrDD	61			X
AllocateCtxHook	63	X		X
FreeCtxHook	64	X		X
ArmCtxHook	65	X	X	X
VMSetMem	66	X		X
OpenEventSem	67	X		
CloseEventSem	68	X		
PostEventSem	69	X		
ResetEventSem	6A	X		
DynamicAPI	6C	X		X

Note 1: The function has limitations executing at interrupt time. See the description of this function for details.

Related DevHlp Services

Related DevHlp services can be grouped together into the following categories:

Advanced BIOS Services

- ABIOSCall (see page 17-12)
- ABIOSCommonEntry (see page 17-13)
- FreeLIDEntry (see page 17-31)
- GetDeviceBlock (see page 17-35)
- GetLIDEntry (see page 17-39).

Character Queue Management

- QueueFlush (see page 17-65)
- QueueInit (see page 17-66)
- QueueRead (see page 17-67)
- QueueWrite (see page 17-68).

PDD-VDD Communications Services

- AttachDD (see page 17-19)
- RegisterPDD (see page 17-72).

Context Hook Services

- AllocateCtxHook (see page 17-14)
- ArmCtxHook (see page 17-18)
- FreeCtxHook (see page 17-29).

Interrupt Management

- EOI (see page 17-28)
- SetIRQ (see page 17-85)
- SetROMVector (see page 17-86)
- UnSetIRQ (see page 17-93).

Memory Management

- AllocGDTSelector (see page 17-15)
- AllocPhys (see page 17-16)
- FreeGDTSelector (see page 17-30)
- FreePhys (see page 17-32)
- LinToGDTSelector (see page 17-41)
- LinToPageList (see page 17-42)
- Lock (see page 17-43)
- PageListToGDTSelector (see page 17-50)
- PageListToLin (see page 17-52)
- PhysToGDTSel (see page 17-54)
- PhysToGDTSelector (see page 17-55)
- PhysToUVirt (see page 17-56)
- PhysToVirt (see page 17-57)
- Unlock (see page 17-91)
- UnPhysToVirt (see page 17-92)
- VerifyAccess (see page 17-94)
- VirtToLin (see page 17-96)

device helper services

- VirtToPhys (see page 17-97)
- VMAlloc (see page 17-98)
- VMFree (see page 17-100)
- VMGlobalToProcess (see page 17-101)
- VMLock (see page 17-103)
- VMProcessToGlobal (see page 17-105)
- VMSetMem (see page 17-107)
- VMUnlock (see page 17-108).

Monitor Management

- DeRegister (see page 17-24)
- MonFlush (see page 17-45)
- MonWrite (see page 17-48)
- MonitorCreate (see page 17-46)
- Register (see page 17-70).

Process Management

- Block (see page 17-21)
- DevDone (see page 17-25)
- RegisterStackUsage (see page 17-73)
- Run (see page 17-78)
- TCYield (see page 17-89)
- Yield (see page 17-109).

Processor Mode Services

- ProtToReal (see page 17-61)
- RealToProt (see page 17-69).

Request Queue Management

- AllocReqPacket (see page 17-17)
- FreeReqPacket (see page 17-33)
- PullParticular (see page 17-62)
- PullReqPacket (see page 17-63)
- PushReqPacket (see page 17-64)
- SortReqPacket (see page 17-88).

Semaphore Management

- CloseEventSem (see page 17-23)
- OpenEventSem (see page 17-49)
- PostEventSem (see page 17-60)
- ResetEventSem (see page 17-75)
- SemClear (see page 17-80)
- SemHandle (see page 17-81)
- SemRequest (see page 17-83).

System Clock Management

- SchedClockAddr (see page 17-79).

System Services

- Beep (see page 17-20)
- SAVE_MESSAGE (see page 17-26)
- DynamicAPI (see page 17-27)
- GetDescInfo (see page 17-34)
- GetDOSVar (see page 17-36)
- InternalError (see page 17-40)
- RegisterBeep (see page 17-71)
- ROMCriticalSection (see page 17-77)
- SendEvent (see page 17-84)
- VideoPause (see page 17-95).

Timer Services

- RegisterTmrDD (see page 17-74)
- ResetTimer (see page 17-76)
- SetTimer (see page 17-87)
- TickCount (see page 17-90).

ABIOSCall

This service is used to invoke an BIOS function for the Operating System Transfer Convention.

Calling Sequence

```
MOV  AX,LID           ; Logical ID
MOV  SI,RB_Offset     ; Offset in data segment to BIOS request block
MOV  DH,Entry_Point   ; Specifies entry point
                        ; 0 = start
                        ; 1 = interrupt
                        ; 2 = timeout
MOV  DL,DevHlp_ABIOSCall
CALL [Device_Help]
```

Results

```
'C' Clear if successful.
    The BIOS service is invoked.

'C' Set if error.
    AX = Error code.
    Possible errors:
        ERROR_INVALID_ENTRY_POINT
        ERROR_BIOS_NOT_PRESENT
        ERROR_NOT_YOUR_LID
        ERROR_LID_DOES_NOT_EXIST
```

Remarks: ABIOSCall sets up the stack for the call to BIOS. The indicated BIOS function is called according to the Operating System Transfer Convention. When the BIOS function returns, ABIOSCall cleans up the stack before returning to the physical device driver.

DS must point to the physical device driver's data segment. If DS was used in a previous call to PhysToVirt, it must be reset to the device driver's data segment.

ABIOSCommonEntry

This service is used to invoke an ABIOS common entry point according to the Advanced BIOS Transfer Convention.

Calling Sequence

```
MOV  SI,RB_Offset      ; Offset in data segment to ABIOS request block
MOV  DH,Entry_Point    ; Specifies entry point
                        ; 0 = start
                        ; 1 = interrupt
                        ; 2 = timeout
MOV  DL,DevHlp_ABIOSCommonEntry
CALL [Device_Help]
```

Results

'C' Clear if successful.
The ABIOS common entry point is invoked.

'C' Set if error.
AX = Error code.
Possible errors:
 ERROR_INVALID_ENTRY_POINT
 ERROR_ABIOS_NOT_PRESENT
 ERROR_NOT_YOUR_LID
 ERROR_LID_DOES_NOT_EXIST

Remarks: ABIOSCommonEntry sets up the stack for the call to one of the Advanced BIOS common entry points. It then invokes the indicated ABIOS common entry point. On return from the ABIOS function, ABIOSCommonEntry cleans up the stack before returning to the physical device driver.

DS must point to the physical device driver's data segment. If DS was used in a previous call to PhysToVirt, it must be reset to the data segment of the physical device driver.

AllocateCtxHook

This service allocates a context hook for use by a physical device driver that needs task time processing, but has no task time thread available to complete it.

Calling Sequence

```
MOV  EAX,Hook_Handler      ; 16 bit offset to context hook handler
MOV  BX,0FFFFFFFFh         ; Reserved value
MOV  DL,DevHlp_AllocateCtxHook

CALL [Device_Help]
```

Results

'C' Clear if hook allocated.
EAX = Hook handle - used by ArmCtxHook.

'C' Set if error.
EAX = Error code.
Possible errors:
Not enough memory
Invalid parameters

Remarks: When the context hook is armed and triggers, the Hook_Handler function is called with the following parameters passed on the call to ArmCtxHook:

EAX = Value passed on the ArmCtxHook DevHelp call in EAX.

EBX = 0FFFFFFFFH, reserved value.

The hook handler is responsible for saving and restoring registers on entry and exit. The hook handler address should be zero-extended, when it is moved into EAX.

DS should be set to the physical device driver's data segment. If the device driver has issued a call to PhysToVirt referencing the DS register, it should restore DS to its original value.

AllocGDTSelector

This service allocates a set of Global Descriptor Table (GDT) selectors for a physical device driver to use. This allocation is performed at device driver INIT (initialization) time.

Calling Sequence

```
MOV  ES,address_high      ; 32-bit address of GDT selector array
MOV  DI,address_low       ;
MOV  CX,number            ; Number of selectors requested
MOV  DL,DevHlp_AllocGDTSelector
```

```
CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

Invalid address

Zero selectors requested

Not enough selectors available

Remarks: AllocGDTSelector is used to allocate a set of GDT selectors for a physical device driver to use for task-time and interrupt-time operations. The address passed in ES:DI gives the location of an array of words to be filled in with the GDT selectors allocated. The value of CX specifies the number of selectors to be allocated. Note that the selector values returned might not be contiguous values. The interrupt handler of a physical device driver must be able to address data buffers, regardless of the context of the current process (the current LDT does not necessarily address the data space that contains the data buffer that the interrupt handler needs to access). PhysToGDTSelector is used to establish the addressability of a GDT selector, and the GDT selector's addressability remains valid and unchanged, until another call to PhysToGDTSelector is made for the same selector.

AllocPhys

This service is used by physical device drivers to allocate a block of fixed memory.

Calling Sequence

```
MOV  BX,size_low           ; Size in bytes
MOV  AX,size_high          ;
MOV  DH,high_or_low        ; Relative position to 1MB
                               ; 0 = above 1MB
                               ; 1 = below 1MB
MOV  DL,DevHlp_AllocPhys
CALL [Device_Help]
```

Results

'C' Clear if the memory was allocated.
AX:BX = 32-bit physical address.

'C' Set if the memory was not allocated.
AX = Error code.
Possible errors:
Memory not allocated

Remarks: The memory allocated by this function is fixed memory, and might not be *unfixed* through the call to the DevHlp,Unlock. If the memory requested is to be allocated *high* (above 1MB) but no memory above 1MB is available, an error is returned. The physical device driver could then attempt to allocate low memory.

Conversely, if the memory requested is to be allocated *low* (below 1MB) but no memory below 1MB is available, an error is returned. The physical device driver could try allocating high memory, if appropriate.

AllocReqPacket

This service returns a pointer to an empty request packet.

Calling Sequence

```
MOV    DH,wait_flag           ; Wait for available request packet
                                   ; 0 = if to wait
                                   ; 1 = if not to wait
MOV    DL,DevHlp_AllocReqPacket
CALL   [Device_Help]
```

Results

'C' Clear if a request packet was allocated.
ES:BX = The virtual address of the allocated request packet.

'C' Set if a request packet was not allocated.

Remarks: Some device drivers, notably the physical Disk device driver, need to have additional request packets to service task-time requests. Request packets that were allocated by AllocReqPacket can be placed in the request packet linked list. Request packets allocated in this manner should be returned to the kernel as soon as possible through the DevHlp, FreeReqPacket. The system has a limited number of request packets, so it is important that a physical device driver not allocate request packets and hold them for future use.

The state of the interrupt flag is not preserved across calls to AllocReqPacket.

ArmCtxHook

This service arms a context hook allocated by the AllocateCtxHook DevHelp service. ArmCtxHook can be called at interrupt time. The next available task-time thread is used to call the function address specified at hook allocation time.

Calling Sequence

```
MOV  EAX,Hook_Data      ; Data to be passed on to the hook handler
MOV  EBX,Hook_Handle    ; handle to the hook to arm
MOV  ECX,0FFFFFFFFh     ; Reserved value
MOV  DL,DevHlp_ArmCtxHook
```

```
CALL [Device_Help]
```

Results

'C' Clear if hook successfully armed.

'C' Set if error.

EAX = Error code.

Possible errors:

Not enough memory

Hook already armed

Invalid parameters

Remarks: After the context hook is armed, it operates once and automatically disarms itself. It is an error to attempt to arm a context hook that is already armed. Once the context hook starts execution, the hook can be rearmed.

The parameter in EAX is passed on to the Hook_Handler routine in the same register.

AttachDD

This service returns the address of the Inter-Device Driver Communication (IDC) entry point to a specified device.

Calling Sequence

```
MOV  BX,OFFSET DS:ddname      ; Name of other device driver
MOV  DI,OFFSET DS:dd@         ; Data area to hold address
MOV  DL,DevHlp_AttachDD

CALL [Device_Help]
```

Results

'C' Clear if no error.
The data area is filled in.

'C' Set if error.
AX = Error code.
Possible errors:
Device driver not found
No IDC entry point

Remarks: The *ddname* field contains the ASCII name of the target device driver. If the target device driver is a character device driver, the *ddname* must match the name in the target device driver's device header.

The *dd@* field must be 12 bytes in length. It has the following layout:

Description	Length
Real Mode Offset of IDC Entry Point	WORD
Real Mode CS Segment of IDC Entry Point	WORD
Real Mode DS of IDC Device Driver	WORD
Protect Mode Offset of IDC Entry Point	WORD
Protect Mode CS Selector of IDC Entry Point	WORD
Protect Mode DS of IDC Device Driver	WORD

Before the physical device driver calls the IDC entry point, the device driver must verify that the entry point it received is non-zero for the CPU mode that calls the IDC entry point. When calling the IDC entry point of the target device driver, the caller must set the DS register for the target device driver. Other registers used in the calling convention must be defined by the target device driver.

The IDC entry point of the target device driver must follow the FAR CALL/RETURN model.

Beep

This service generates a beep.

Calling Sequence

```
MOV  BX,usFreq      ; Frequency of beep in Hz
MOV  CX,usDuration   ; Duration of beep in milliseconds
MOV  DL,DevHlp_Beep

CALL [Device_Help]
```

Results

'C' Clear if successful.
AX = 0.

'C' Set if error.
AX = Error code.
Possible errors:
Invalid frequency

Remarks: This function uses the DX register.

Block

This service prevents the thread executing in the physical device driver from executing (“puts it to sleep”) until either a call to the DevHlp, Run, is issued on the event identifier, or a timeout occurs.

Calling Sequence

```
MOV  BX,event_id_low      ; Low word of event id
MOV  AX,event_id_high     ; High word of event id
MOV  DI,time_limit_high   ; Timeout interval in milliseconds
MOV  CX,time_limit_low    ; ( = -1 if to never timeout)
MOV  DH,interruptible_flag ; Tells if sleep is interruptible
                                ; 0 = interruptible
                                ; 1 = non-interruptible

MOV  DL,DevHlp_Block

CALL [Device_Help]
```

Results

```
'C' Clear if event wakeup.
'C' Set if unusual wakeup.
'Z' Set if wakeup due to timeout.
'Z' Clear if sleep was interrupted.
```

AL = Awake code, non-zero if unusual wakeup.

Remarks: The return from the call to Block indicates whether the “wakeup” occurred as the result of a call to Run, or an expiration of the time limit. Block removes the current thread from the Run queue, and starts executing some other thread. The thread blocked in the physical device driver is reactivated and Block returns, when Run is called with the same event identifier, when the time limit expires, or when the thread is signalled.

The event identifier is an arbitrary 32-bit value, but a convention must be followed to coordinate with the thread issuing the Run function. The standard convention for Block/Run operations is to use the address of some structure or memory cell associated with the reason for blocking and running. For example, a thread blocking until some resource is cleared normally blocks on, the address of the ownership flag for that resource.

The Block/Run mechanism is so designed that it cannot guarantee immunity from a faulty wakeup by some other thread in the system running a 32-bit key value (which happens to match the key of some unrelated blocking thread). The goal is to choose keys that have a high likelihood of being unique. However, users of Block/Run must always check the reason for the wakeup to make sure that the event actually took place, and that the wakeup was not accidental.

When calling Block, it is important to use the sequence:

Disable Interrupts

```
while (need to wait)
    Block (value)
Disable Interrupts
```

Interrupts must be turned off before checking the condition (for example, I/O done, resource freed). This is necessary to avoid a deadlock caused by getting an interrupt-time Run call before completing the call to Block. Block re-enables the interrupts. Note the use of the *WHILE* clause. It is essential to recheck the awaited condition and, if necessary, re-disable interrupts and call Block again. The convention of using an address as an event identifier should prevent double use of an identifier. A time limit of - 1 means that

Block waits indefinitely until Run is called. Block can be called only by the task-time portion of a physical device driver.

When using Block to block a thread, the driver can specify whether the *sleep* can be interrupted. If the sleep is interruptible, then the kernel can abort the blocked thread and return from the Block, without using a corresponding Run. In general, the sleep should be marked as interruptible, unless the sleep duration is expected to be short; that is, less than a second.

If the return from the Block indicates that the sleep was interrupted, some internal event occurred that requires attention (such as a signal, process death, or some other forced action). The device driver should respond by performing any necessary cleanup, setting the error code in the status field of the request packet, setting the done bit, and returning the request packet to the kernel.

CloseEventSem

This service closes an event semaphore that was previously opened with OpenEventSem. If this is the last reference to this event, then the event semaphore is destroyed.

Calling Sequence

```
MOV  EAX, SemaphoreHandle      ; DWORD semaphore handle
MOV  DL, DevHlp_CloseEventSemaphore

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

EAX = Error code.

Possible errors:

Invalid handle

Remarks: This function uses the EAX and Flags registers. CloseEventSem can be called only from a Ring 0 device driver, or file system driver. The handle *passed in* must be a handle to a *shared event* semaphore. If the handle does not exist or is not a shared event semaphore, or if the semaphore was not previously opened with OpenEventSem, then ERROR_INVALID_HANDLE is returned.

The system semaphores reside in a memory buffer rather than on a disk file. Therefore, when the last process that has a semaphore open, exits or closes that semaphore, the semaphore disappears.

The OPEN/CLOSE operations can be nested. A maximum of 65,534 (64KB – 1) opens-per-process is allowed for each semaphore at any one time. If this limit is reached, the very next call to OpenEventSem returns ERROR_TOO_MANY_OPENS. In order for a process to intentionally destroy a semaphore prior to termination, the number of calls to CloseEventSem must equal the number of calls to OpenEventSem.

DeRegister

This service removes all of the monitors associated with the specified task from the specified monitor chain.

Calling Sequence

```
MOV  BX,monitor_PID      ; Process ID of monitor task
MOV  AX,monitor_handle    ; MonitorCreate handle for chain
MOV  DL,DevHlp_DeRegister
```

```
CALL [Device_Help]
```

Results

'C' Clear if successful.

AX = The number of monitors still registered in the chain, after deregistration.

'C' Set if error.

AX = Error code.

Possible errors:

Invalid monitor handle

Invalid parameter

Remarks: This function can be called only at task time in OS/2 mode. To remove a monitor from a monitor chain, the physical device driver must supply the Process ID (PID) of the process that owns the monitor being removed, and the handle of the monitor chain that is affected. All monitors belonging to the specified PID are removed from the specified monitor chain. A single process can register more than one monitor with the same monitor chain. Therefore, DeRegister removes all monitors associated with the specified process from the specified monitor chain.

DevDone

This service signifies that a request has completed and unblocks any threads waiting in the kernel for the request.

Calling Sequence

```
LES  BX,request_packet    ; Pointer to I/O request packet
MOV  DL,DevHlp_DevDone

CALL [Device_Help]
```

Results: None.

Remarks: DevDone sets the done bit in the Status field of the request packet header and issues the Run DevHlp service for threads, which are blocked in the kernel waiting for the request packet to be completed. DevDone is called from the device interrupt routine. The physical device driver that issues the DevDone command should set any error flags in the Status field before calling the routine. DevDone does not apply to request packets that were allocated from the DevHlp, AllocReqPacket.

The physical device driver does not call DevDone for requests that are completed at task time (in the strategy routine). Requests that are completed at strategy time should return with the done bit set in the request packet.

SAVE_MESSAGE

This service displays a message from a base physical device driver on the system console.

Calling Sequence

```
MOV  DS,SEG  Msg_Table    ; DS = Segment of message table
MOV  SI,OFFSET Msg_Table  ; SI = Offset of message table
XOR  BX,BX                ; BX = Reserved (must be 0)
MOV  DL,DevHlp_SAVE_MESSAGE
```

```
CALL [Device_Help]
```

Results: None

Remarks: The message is not displayed immediately, but is queued until system initialization retrieves it from the system message file.

The structure of the message table is:

WORD	Message ID
WORD	Number of fill-in items
DWORD	Pointer to first fill-in item of ASCIIIZ string
DWORD	Pointer to second fill-in item of ASCIIIZ string
.	.
.	.
DWORD	Pointer to last fill-in item of ASCIIIZ string

DynamicAPI

This service allows a physical device driver to dynamically create a call gate to a Ring 0 service (worker function) contained within the physical device driver.

Calling Sequence

```

MOV  BX, Worker_Low    ; If 16:16 address, ax = selector
MOV  AX, Worker_High    ;                bx = offset
                        ; If 32-bit linear, ax = high order
                        ;                bx = low order

MOV  CX, Parm_Count     ; If 16-bit call gate, cx = WORD count
                        ; If 32-bit call gate, cx = DWORD count

MOV  DH, Flag           ; Bit 0 (mask =1) on = 16-bit call gate
                        ;                off = 32-bit call gate
                        ; Bit 1 (mask =2) on = 16:16 worker address
                        ;                off = linear worker address

MOV  DL, DevHlp_DynamicAPI

CALL [Device_Help]

```

Results

'C' Clear if successful.
DI = Callgate selector.

'C' Set if error.
EAX = Error code.

Remarks: The function generates an indirect call to the worker through the returned call gate. The worker routine is given control in kernel mode. Exit kernel mode is performed upon returning from the worker. The worker routine receives control with SI containing the offset into the stack, where the user parameters have been copied. If the API worker returns with the carry flag clear, AX/EAX is set to 0, prior to returning to the calling application. If the carry flag is set, an error code might be returned in AX.

Parm_count cannot exceed 16, for 16-bit *call gates*, and cannot exceed 8, for 32-bit *call gates*.

A 16:16 worker routine returns to the kernel with a RETF instruction. A 32-bit linear worker routine returns with a RETFD instruction.

EOI

This service is used to issue an End-Of-Interrupt (EOI) to the master/slave 8259 interrupt controllers as appropriate to the interrupt level.

Calling Sequence

```
MOV  AL,IRQnum          ; Interrupt level number (0-F)
MOV  DL,DevHlp_EOI
CALL [Device_Help]
```

Results: None.

Remarks: This function is used to issue an End-Of-Interrupt to the 8259 interrupt controllers for a physical device driver interrupt handler. If the specified interrupt level is for the slave 8259 interrupt controller, then this service issues the EOI to both the master and slave 8259s. Physical device drivers must use this service in their interrupt handlers for upward compatibility.

This DevHlp does not change the state of the interrupt flag. If the physical device driver returns to the operating system immediately after issuing the EOI, then it should disable interrupts prior to the EOI. Disabling interrupts prior to issuing the EOI allows the processing for this interrupt level to be completed before the system services the next interrupt. This reduces the risk of excessive nested interrupts causing a system stack overflow. If any post-EOI work is done by the interrupt handler, it should be limited to the first or non-nested interrupt. Nested interrupt processing should be done only prior to the EOI.

Notice that this routine can be called at initialization time for interrupt processing.

FreeCtxHook

This service frees a context hook allocated by the AllocateCtxHook DevHelp service.

Calling Sequence

```
MOV  EAX,Hook_Handle      ; Context hook handle
MOV  DL,DevHlp_FreeCtxHook

CALL [Device_Help]
```

Results

'C' Clear if hook freed.
EAX = 0.

'C' Set if error.
EAX = Error code.

FreeGDTSelector

This service frees a selector allocated with the AllocateGDTSelector DevHlp service.

Calling Sequence

```
MOV  AX,Selector          ; GDT selector to free
                        ;   (allocated using AllocateGDTSelector).
MOV  DL,DevHlp_FreeGDTSelector
CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

EAX = Error code.

Possible errors:

ERROR_ACCESS_DENIED

ERROR_INVALID_PARAMETER

Remarks: FreeGDTSelector can block, so it is not available at interrupt time. This function uses AX and DL. The selector passed to this function must have been allocated using AllocateGDTSelector. This will be verified and an error returned, if the selector was not properly allocated.

FreeLIDEntry

This service is used to release a Logical ID (LID). This must be done at deinstall or termination time.

Calling Sequence

```
MOV  AX,LID           ; Logical ID from GetLIDEntry
MOV  DL,DevHlp_FreeLIDEntry
CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

ERROR_NOT_YOUR_LID

ERROR_LID_DOES_NOT_EXIST

ERROR_ABIOS_NOT_PRESENT

Remarks: The attempt to free a Logical ID can fail, if the physical device driver does not own the LID, or if the LID does not exist.

DS must point to the data segment of the physical device driver. If DS was previously used in a call to PhysToVirt, it must be reset to the data segment of the physical device driver.

FreePhys

This service is used to release physical memory allocated by the DevHlp, AllocPhys.

Calling Sequence

```
MOV  BX,address_low      ; 32-bit physical address
MOV  AX,address_high     ;
MOV  DL,DevHlp_FreePhys

CALL [Device_Help]
```

Results

'C' Clear if memory freed.
'C' Set if memory not freed.

Remarks: Any memory that the physical device driver allocated by way of the AllocPhys should be released prior to device driver termination. Memory that was not allocated with AllocPhys cannot be freed.

FreeReqPacket

This service is used to release a request packet previously allocated by AllocReqPacket.

Calling Sequence

```
LES  BX,request_packet      ; Pointer to request packet previously allocated
MOV  DL,DevHlp_FreeReqPacket

CALL [Device_Help]
```

Results: None.

Remarks: The physical device driver should only free request packets that were previously allocated by AllocReqPacket. The Device Helper service, DevDone, should not be used to return an allocated request packet. The system has a limited number of request packets, so it is important that a physical device driver not allocate request packets and hold them for future use.

The state of the interrupt flag is not preserved across calls to this DevHlp.

GetDescInfo

This service is used to obtain information about the contents of a descriptor.

Calling Sequence

```
MOV  AX,Selector          ; The selector referring to the descriptor
MOV  DL,DevHlp_GetDescInfo

CALL [Device_Help]
```

Results

'C' Clear if successful.

- AL = Descriptor's access byte.
- AH = If the descriptor is not a call gate,
AH = The Big and Granularity fields of attribute byte.
If the descriptor is a call gate,
AH = The number of parameters.
- ECX = If the descriptor is not a call gate,
ECX = 32 bit linear address stored in descriptor.
If the descriptor is a call gate,
CX = Selector.
- EDX = If the descriptor is not a call gate,
EDX = The 32-bit, byte-granular size of descriptor (=0 if 4 GB).
If the descriptor is a call gate,
EDX = 32-bit offset (0:32 addressing).

'C' Set if error.

- EAX = Error code.
Possible errors:
Descriptor invalid

Remarks: When called for an Local Descriptor Table (LDT) descriptor, GetDescInfo can block other threads from executing. Therefore, at interrupt time, this routine is callable only on Global Descriptor Table (GDT) descriptors. The routine can be called with either type of descriptor at initialization or task time.

GetDeviceBlock

This service returns an ABIOS device block pointer. This function returns a protect-mode pointer only. Real-mode pointers are not returned, instead, the data is initialized to 0.

Calling Sequence

```
MOV  AX,LogicalID
MOV  DS,Data_Segment      ; Data segment of the requesting device driver
MOV  DL,DevHlp_GetDeviceBlock
```

```
CALL [Device_Help]
```

Results

'C' Clear if successful.
CX:DX = Protect mode device block pointer.
AX:BX = 00:00
(was real mode device block pointer; returned as 00:00 for compatibility).

'C' Set if error.
AX = Error code.
Possible errors:
 ABIOS not present
 Not your LID
 LID does not exist

Remarks: Except for the Exit registers, all registers are unaltered.

GetDOSVar

This service is used to return the address of a kernel variable.

Calling Sequence

```
MOV  AL,index          ; Index wanted.
MOV  DL,DevHlp_GetDOSVar
CALL [Device_Help]
```

Results

'C' Clear if successful.
AX:BX points to the index.

'C' Set if error.

Remarks: The following table contains the list of *read only* variables:

Index	Variable Description
1	GlobalInfoSeg:WORD. Valid at task time and interrupt time, but not at INIT time. See below.
2	LocalInfoSeg:DWORD. Selector/segment address of local information segment for the current Local Descriptor Table (LDT). Valid only at task time. See below.
3	Reserved.
4	VectorSDF:DWORD. Pointer to the stand-alone dump facility. Valid at task time and interrupt time.
5	VectorReboot:DWORD. Pointer to restart the operating system. Valid at task time and interrupt time.
6	Reserved.
7	YieldFlag:BYTE. Indicator for performing yields. Valid only at task time.
8	TCYieldFlag:BYTE. Indicator for performing time-critical yields. Valid only at task time.
9	Reserved.
10	Reserved.
11	DOS Session Code Page Tag pointer: DWORD. Segment:offset of the DOS Session's current code page tag. Valid only at task time.

GlobalInfoSeg (PSEL)	Address of the global information segment structure, as defined below:	
	Time (ULONG)	Time in seconds since 1/1/1970.
	MilliSecs (ULONG)	Time in milliseconds.
	Hours (UCHAR)	Current hour.
	Minute (UCHAR)	Current minute.
	Seconds (UCHAR)	Current second.
	HundredSec (UCHAR)	Current hundredth of a second.
	TimeZone (USHORT)	Minutes from UTC. If FFFFH, TimeZone is undefined.
	Interval (USHORT)	Timer interval in tenths of milliseconds.
	Day (UCHAR)	Day.
	Month (UCHAR)	Month.

Year (USHORT)	Year.
Weekday (UCHAR)	Day of the week: 0 Sunday 1 Monday 2 Tuesday 3 Wednesday 4 Thursday 5 Friday 6 Saturday.
MajorVersion (UCHAR)	Major version number.
MinorVersion (UCHAR)	Minor version number.
Revision (UCHAR)	Revision letter.
CurrentSession (UCHAR)	Current foreground full-screen session.
MaxNumSessions (UCHAR)	Maximum number of full-screen sessions.
HugeShift (UCHAR)	Shift count for huge segments.
ProtModelnd (UCHAR)	Protect-mode-only indicator: 0 DOS mode and OS/2 mode 1 OS/2 mode only.
LastProcess (USHORT)	Process ID of the current foreground process.
DynVarFlag (UCHAR)	Dynamic variation flag: 0 Absolute 1 Enabled.
MaxWait (UCHAR)	Maximum wait in seconds.
MinTimeSlice (USHORT)	Minimum timeslice in milliseconds.
MaxTimeSlice (USHORT)	Maximum timeslice in milliseconds.
BootDrive (USHORT)	Drive from which the system startup occurred: 1 Drive A 2 Drive B • • • • • • n Drive <i>n</i> .
TraceFlags (UCHAR)	Thirty-two system trace major code flags. Each bit corresponds to a trace major code, 00H – FFH. The most significant bit (left-most) of the first byte corresponds to major code, 00H. Values are: 0 Trace disabled 1 Trace enabled.
MaxTextSessions (UCHAR)	Maximum number of VIO windowable sessions.
MaxPMSessions (UCHAR)	Maximum number of Presentation Manager sessions.
LocalInfoSeg (PSEL)	Address of the selector for the local information segment structure, as defined below:
ProcessID (PID)	Current Process ID.

ParentProcessID (PID)	Parent Process ID.
ThreadPrty (USHORT)	Priority of current thread.
ThreadID (TID)	Current Thread ID.
SessionID (USHORT)	Current Session ID.
ProcStatus (UCHAR)	Process status.
Unused (UCHAR)	Unused.
ForegroundProcess (BOOL)	Current process is in foreground (has keyboard focus): -1 Implies <i>yes</i> 0 Implies <i>no</i> .
TypeProcess (UCHAR)	Type of process: 0 Full screen protect-mode session 1 Requires real mode 2 VIO windowable protect-mode session 3 Presentation Manager protect-mode session 4 Detached protect-mode process.
Unused (UCHAR)	Unused.
EnvironmentSel (SEL)	Environment selector.
CmdLineOff (USHORT)	Command line offset in the segment addressed by EnvironmentSel.
DataSegLen (USHORT)	Length of the data segment in bytes.
StackSize (USHORT)	Stack size in bytes.
HeapSize (USHORT)	Heap size in bytes.
HModule (UHMODULE)	Module handle.
DSSel (SEL)	Data segment selector.

These variables are maintained by the kernel for the benefit of physical device drivers. Notice that the address returned is the address of the indicated variable; the variable can contain a vector to some facility, or it can contain some structure.

GetLIDEntry

This service is used to obtain a Logical ID (LID) for devices that exist, that is, devices that are *awake*.

Calling Sequence

```
MOV  AL,DeviceID          ; Desired Device ID
MOV  BL,RelativeLID#      ; Nth Logical ID of this Device ID (0 - FF) where
                          ; 0 = first unclaimed LID (that is, first one available)
                          ; 1 = the first LID
MOV  DH,DeviceState       ; Requested LID indicator
                          ; 0 = all other LIDs
                          ; 1 = DMA, POS
MOV  DL,DevHlp_GetLIDEntry
CALL [Device_Help]
```

Results

'C' Clear if successful.
AX = LID number.

'C' Set if error.
AX = Error code.
Possible errors:
ERROR_LID_ALREADY_OWNED
ERROR_LID_DOES_NOT_EXIST
ERROR_ABIOS_NOT_PRESENT

Remarks: This function is used by a physical device driver to obtain a LID entry. Because OS/2 2.0 does not support the BIOS Sleep/Wake functions, only devices that are “awake” are considered to exist, and thus are available to device drivers. The physical device driver can use this service in two ways:

- By specifying a relative LID. Because the ordering of LIDs corresponds to the ordering of the physical devices, a physical device driver that supports a certain relative device can determine if an LID entry is available. An example would be a character device driver that supports COM4; the physical device driver would need to get the LID entry for the fourth COM port.
- By requesting the first available LID for its device type. An example of this is a block device driver that wants to get the first available LID for diskettes.

GetLIDEntry searches the BIOS Common Data Area table for an entry corresponding to the specified Device ID. If an entry is located that matches the caller's form of request, it is returned to the caller. If a LID entry is found but already owned, an error is returned. If no LID entry is found, an error is also returned.

Some LIDs are not allocated to device drivers. Certain Device IDs are used by the operating system kernel to perform such actions as mode switching. The reserved Device ID is for System Services.

Certain LIDs are allocated as shared. For these Device IDs, GetLIDEntry allows multiple device drivers to access the LID concurrently. It is up to the physical device driver to determine if the device is busy, or available for use when needed. The two Device IDs that are allocated as shared are DMA and POS.

DS must point to the data segment of the physical device driver. If DS was previously used in a call to PhysToVirt, it must be reset to the data segment of the physical device driver.

Note: GetLIDEntry must be called with the DeviceState parameter set to 1, in order to obtain a LID for these Device IDs. In all other cases, DeviceState must be set to 0.

InternalError

This service is called when an internal inconsistency has been detected.

Calling Sequence

```
LDS  SI, message_text
MOV  DI, message_text_length
MOV  DL, DevHlp_InternalError
CALL [Device_Help]
```

Results: This routine does not return to the caller.

Remarks: This function is used only when an internal inconsistency is detected. In this situation, the system might be unable to continue safely. This should be used only when the operation of the system is questionable. A minor device driver (Parallel Port, COM, or other) might never use this routine.

The maximum message length is 128 characters. Longer messages are truncated. The message display code is limited. The message should contain ASCII characters in the range 0x20 through 0x7E, with optional 0x0D and 0x0A to break the text into multiple lines. The physical device driver name must preface the message text.

LinToGDTSelector

This service converts a linear address to a virtual (selector:offset) address by mapping the given Global Descriptor Table (GDT) selector to the memory region referred to by the given linear address and range. The size of the range mapped must be less than, or equal to, 64KB.

Calling Sequence

```
MOV  AX,Selector      ; GDT selector to map (allocated using AllocateGDTSelector)
MOV  EBX,Linear_Address ; FLAT address of the memory region to be mapped;
                        ; must be on a page boundary
MOV  ECX,Size         ; Size of the range to be mapped, in bytes;
                        ; must be less than or equal to 64 KB
MOV  DL,DevHlp_LinToGDTSelector
CALL [Device_Help]
```

Results

```
'C' Clear if successful.

'C' Set if error.
  EAX = Error code.
    Possible errors:
      ERROR_INVALID_PARAMETER
```

Remarks: The memory that is being mapped must be fixed or locked prior to a call to this function. After this function is issued for a particular selector, the addressability remains valid, until the physical device driver changes its content with a subsequent call to the DevHlp services, PageListToGDTSelector, PhysToGDTSel, PhysToGDTSelector, or LinToGDTSelector.

LinToPageList

This service translates a linear address range to an array of PageList_s structures that describes the physical pages to be mapped.

Calling Sequence

```
MOV  EAX,LAddr          ; Starting linear address of the range to be translated
                          ;   into a PageList array
MOV  ECX,cbSize          ; Size of the range to translate in bytes;
                          ;   must be less than or equal to 64KB
MOV  EDI,OFFSET pPageList ; Flat pointer to the array or PageList_s structures that
                          ;   will be filled in by the call
MOV  DL,DevHip_LinToPageList
CALL [Device_Help]
```

The linear address range is translated into an array of PageList_s structures. Each PageList_s structure describes a single physically contiguous subregion of the physical memory that is mapped by the linear range. The format of the PageList_s structure is:

```
PageList_s struc
    pl_PhysAddr DD ? ; Physical address of first byte in this sub-region
    pl_cb       DD ? ; Number of contiguous bytes starting at pl_PhysAddr
PageList_s ends
```

The sum of the *pl_cb* fields in the PageList array produced by this function is equal to *cbSize*.

Results

'C' Clear if successful.
 pPageList will contain the physical mapping of the range.
 EAX = Number of elements in PageList array.

'C' Set if error.

Remarks: The physical pages that are mapped by the linear range must be fixed or locked prior to this call. It is the responsibility of the device driver to ensure that enough entries have been reserved for the range of memory being translated (possibly one entry per page in the range plus one more, if the region does not begin on a page boundary).

Lock

This service is called by physical device drivers at task time to lock a memory segment. If the segment is unavailable, the caller must specify whether Lock should block execution until the segment is available, or should return immediately.

Calling Sequence

```

MOV  AX,segment_@      ; Selector or segment
MOV  BL,waitflag        ; 0 = Block until available
                        ; 1 = Return if it is not immediately available
MOV  BH,typeflag        ; Type of lock
                        ; Bit 0 (duration):
                        ;   0 = if short-term
                        ;   1 = if long-term
                        ; Bit 1 (where)
                        ;   0 = if any memory
                        ;   1 = if high memory
                        ; Bit 2 (type)
                        ;   0 = if normal lock
                        ;   1 = if verify lock

MOV  DL,DevHlp_Lock

CALL [Device_Help]
```

Results

'C' Clear if segment locked.
AX:BX = lock handle.

'C' Set if segment unavailable or invalid handle.

Note: AX and BX are not preserved.

Remarks: Only the following combinations of type flags are valid:

00H = Short-term, any memory. Segment is marked fixed at its current physical address.

01H = Long-term, any memory. Segment is marked fixed, and the system can move it into the region reserved for fixed segments. Thus, the physical address of the segment can be changed by the Lock call.

03H = Long-term, high memory. Segment is marked fixed, and the system can move it into the region reserved for fixed segments. Thus, the physical address of the segment can be changed by the Lock call. If the Lock succeeds, the segment is guaranteed to be in high memory. This type is available only at initialization time. No lock handle is returned, and the selector must be that of a segment from the load image of the physical device driver. This lock is irreversible.

04H = Short term, any memory, verify lock. Segment is not made present, and remains swappable. It is not freed or shrunk until the verify lock is removed. Blocking can be performed, when accessing a segment that has been verify locked, but valid access is guaranteed.

In the event of a failure, a long-term lock will return even if the caller specified the request, Block until available..

The duration of the lock must be set to long-term, or a verify lock should be used, for operations that are expected to take two or more seconds to complete. Use of short-term locks for longer periods of time can prevent an adequate amount of movable, swappable memory from being available for system use, and *cause a system halt*. A physical device driver can use short-term locks where necessary for its own code and data segments, provided the 2-second rule is not violated.

DevHlp_Lock

Lock need be done only on segments that have not been locked already by the OS/2 kernel (for example, an address that is passed to the physical device driver through an IOCTL). If the physical device driver plans to lock the segment down, it should call Lock before calling VerifyAccess.

MonFlush

This service removes all data from a specified monitor chain, such as the data stream.

Calling Sequence

```
MOV  AX,monitor_handle      ; MonitorCreate handle for chain
MOV  DL,DevHlp_MonFlush

CALL [Device_Help]
```

Results

'C' Clear if successful.
AX = 0.

'C' Set if error.
AX = Error code.
Possible errors:
Invalid monitor handle
Invalid parameters

Remarks: This service can be called at task time in the OS/2 mode only. If this function is called in a DOS Session context, an invalid parameter error is returned to the physical device driver. When a physical device driver calls MonFlush, the OS/2 Monitor Dispatcher creates and places a *flush record* into the monitor chain. The general format of monitor records requires that every record contain a flag WORD as the first entry. One of the flags is used to indicate that this record is a flush record. The flush record consists only of the flag WORD. This record is used by monitors along the chain to reset internal state information, and to assure that all internal buffers are flushed. The flush record must be passed along to the next monitor because the monitor dispatcher will not process any more information until the flush record is received at the end of the monitor chain; that is, it is returned to the physical device driver's monitor chain buffer at the end of the monitor chain.

Note: Subsequent MonWrite requests fail (or block) until the flush completes; that is, until the flush record is returned to the device driver's monitor chain buffer.

The state of the interrupt flag is not preserved across calls to this DevHlp service.

MonitorCreate

This service creates an empty chain of monitors, or removes an empty chain of monitors.

Calling Sequence

```
LES  SI,final_buffer    ; Address of device driver's monitor chain buffer
LDS  DI,notify_rtn      ; Address of notification routine
MOV  AX,Handle          ; Handle for this chain
                        ; 0 = create new monitor chain
                        ; !0 = specifies chain to be removed
                        ;      (returned from previous create call)
MOV  DL,DevHlp_MonitorCreate
CALL [Device_Help]
```

Results

'C' Clear if successful.
 AX = Monitor chain handle, if Handle was "0."
 AX = 0, if handle was not "0."

'C' Set if error.
 AX = Error code.
 Possible errors:
 Invalid monitor handle
 Not enough memory
 Monitor chain not empty
 Invalid parameter

Remarks: This service can be called only at task time in the OS/2 mode. If this function is called in a DOS Session context, an invalid parameter error is returned to the physical device driver.

The monitor chain buffer, *final_buffer*, is owned by the physical device driver. On calling MonitorCreate, the first WORD of this buffer is the length of the buffer in bytes (including the first WORD). When the monitor chain handle specified is 0, a new monitor chain is created. When the monitor chain handle specified is a handle that was previously returned from a call to MonitorCreate (that is, the handle does not equal 0), the monitor chain referenced by that handle is destroyed.

A *monitor chain* is a list of monitors, with a device driver monitor chain buffer address and code address as the last element on this list. Data is placed into a monitor chain through the MonWrite function. The monitor dispatcher feeds the data through all registered monitors, putting the resulting data, if any, into the specified device driver monitor chain buffer. When data is placed in this buffer, the physical device driver's notification routine is called at task time. The physical device driver should initiate any necessary action in a timely fashion and return from the notification entry point without delay.

Note: If MonWrite is called at interrupt time and the monitor chain is empty, the device driver notification routine is called at interrupt time. Under all other circumstances, it is called at task time.

MonitorCreate establishes one of these monitor chains. The chains are created empty so that data written into them is placed immediately into the buffer of the physical device driver. This service can also destroy a monitor chain, if the handle parameter (AX) is non-zero. The non-zero value is the handle of the chain to remove. If the monitor chain to be removed is not empty (that is, all monitors registered with this chain have not been previously deregistered), an invalid parameter error is returned to the physical device driver.

A call to MonitorCreate must be made before a monitor can be registered (using the Register DevHlp service) with the chain. This can be done at any time, including during the installation of the device driver at system initialization.

The following are notification routine considerations:

- The physical device driver's notification routine, *notify_rtn*, is called by the monitor dispatcher when a data record has been placed in the monitor chain buffer of the device driver. The monitor dispatcher sets ES:SI to the address of the device driver's monitor chain buffer, and DS to the physical device driver's DS, before calling the notification routine.
- The physical device driver must process the contents of the monitor chain buffer before returning to the monitor dispatcher. This entry point is called in the OS/2 mode only.
- When *notify_rtn* is called, the first WORD of the buffer is filled in with the length of the record just sent to the device driver. There is one notification routine call for each record.
- *final_buffer* must reside within the first data segment of the physical device driver, that is, the physical device driver's header segment. *notify_rtn* must reside in the first code segment of the physical device driver.

MonWrite

This service passes data records to the monitors for filtering.

Calling Sequence

```

LDS  SI,data_record_offset    ; Offset of data record in DS
MOV  CX,count                 ; Byte count of data record
MOV  AX,monitor_handle        ; Handle for chain returned from
                              ; Previous MonitorCreate call
MOV  DI,millisecond_high      ; High word of milliseconds
MOV  BX,millisecond_low       ; Low word of milliseconds
MOV  DH,wait_flag             ; Wait/No-Wait/WAIT_TIMEOUT flag
MOV  DL,DevHlp_MonWrite
CALL [Device_Help]

```

Results

'C' Clear if successful.
AX = 0.

'C' Set if error.
AX = Error code.
Possible errors:
Invalid monitor handle
Not enough memory
Call interrupted
Error semaphore timeout

Remarks: MonWrite can be called at task time or interrupt time, in either the OS/2 session or DOS Session. Wait_flag is set to 0, if the MonWrite request occurs at task or user time, and the device driver indicates that the monitor dispatcher does the synchronization. That is, the physical device driver waits until the data can be placed into the monitor chain before the monitor dispatcher returns to the physical device driver. If wait_flag is set to 1, the physical device driver does not wait, and if the data cannot be placed into the monitor chain, the monitor dispatcher returns immediately with the appropriate error. Wait_flag must be set to 1, if the MonWrite request occurs at interrupt time. Wait_flag is set to 2, if the MonWrite request occurs at task or user time, and the physical device driver indicates that the monitor dispatcher will do the synchronization for the time (in milliseconds) specified in DI:BX.

The DS register must be set to the data segment of the physical device driver.

The error, ERROR_NOT_ENOUGH_MEMORY, is returned to the physical device driver when the MonWrite call is made, and the monitors are not able to receive the data. If this condition occurs at interrupt time, an overrun occurred. If it occurs at task (or user) time, the process can block. This error is also returned to the physical device driver when a flush record, sent to the monitors by a previous MonFlush call, is not returned to the physical device driver.

If the thread, on which the physical device driver calls MonWrite, blocks, and is awakened because the process that owns the thread is terminating, a call-interrupted error is returned to the physical device driver. The physical device driver must return the error to the calling process so that the process can complete its termination processes.

Each call to MonWrite sends a single complete record. The data sent by this call is considered to be a complete record. A data record must not be longer than two bytes less the length of the device driver's monitor chain buffer.

The state of the interrupt flag is not preserved across calls to DevHlp.

OpenEventSem

This service opens a 32-bit shared event semaphore.

Calling Sequence

```
MOV  EAX, SemaphoreHandle    ; DWORD semaphore handle
MOV  DL, DevHlp_OpenEventSem

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

EAX = Error code.

Possible errors:

Invalid handle

Out of memory

Too many opens

Remarks: This function uses the EAX and FLAGS registers. OpenEventSem can be called only from a Ring 0 physical device driver, or file system driver. The handle passed in must be a handle to a *shared event* semaphore. If the handle does not exist, or is not a shared event semaphore, then ERROR_INVALID_HANDLE is returned.

The OPEN/CLOSE operations can be nested. A maximum of 65,534 (64KB–1) opens-per-process are allowed for each semaphore at any one time. If this limit is reached, the next call to OpenEventSem returns ERROR_TOO_MANY_OPENS. In order for a process to intentionally destroy a semaphore prior to termination, the number of calls to CloseEventSem must equal the number of calls to OpenEventSem.

PageListToGDTSelector

This service maps physical addresses described in an array of `PageList_s` structures to a GDT (Global Descriptor Table) selector, setting the access byte of the descriptor to the requested type. The virtual memory needed to map the physical ranges described by the `PageList` array must not exceed 64KB.

Calling Sequence

```
MOV  AX,Selector      ; GDT selector to map; allocated with AllocateGDTSelector
MOV  ECX,cbSize       ; Count of bytes to be mapped;
                        ; sum of the pl_cb fields in the PageList array
MOV  EDI,OFFSET pPageList ; Flat pointer to an array of PageList_s
                        ; structures to map the selector to.
MOV  DH, Access       ; Descriptor's type and privilege level
                        ; = 0 The selector is mapped as Ring 3, readable code with
                        ;     16-bit addressing/operand size (286 format).
                        ; = 1 (binary 0001) The selector is mapped as Ring 3,
                        ;     writable data.
                        ; = 3 (binary 0011) The selector is mapped as Ring 2, IOPL,
                        ;     readable code with 16 bit addressing/operand size.
                        ; = 4 (binary 0100) The selector is mapped as Ring 2, IOPL,
                        ;     writable data.
                        ; = 5 (binary 0101) The selector is mapped as Ring 0,
                        ;     readable code with 16 bit addressing/operand size.
                        ; = 6 (binary 0110) The selector is mapped as Ring 0,
                        ;     writable data.
                        ; In addition, 128 (binary 10000000) can be OR'ed onto any of
                        ; the above access values and will cause the selector to
                        ; be mapped with 32 bit default addressing/operand size
                        ; (the 'B' or 'D' bit will be set in the selector).
                        ; Any other values are invalid.
```

```
MOV  DL,DevHlp_PageListToGDTSelector
```

```
CALL [Device_Help]
```

`pPageList` is the flat address of an array of `PageList_s` structures. Each `PageList_s` structure describes a single physically contiguous subregion of the physical memory to be mapped. The format of the `PageList_s` structure is:

```
PageList_s struc
    pl_PhysAddr DD ? ; Physical address of first byte
                    ; in this sub-region
    pl_cb      DD ? ; Number of contiguous bytes
                    ; starting at pl_PhysAddr
PageList_s ends
```

Results

'C' Clear if pages locked.
AX = Selector with the modified RPL (Requested Privilege Level) bits.

'C' Set if segment unavailable.
EAX = Error code.

Remarks: The physical memory that is being mapped must be fixed or locked prior to this call. After this call, offset 0 within the selector corresponds to the first byte in the first entry in the array pointed to by pPageList. If the PageList is an unmodified return array from VMLock or LinToPageList, then the mapping returned from this call is, byte for byte, the same as the original linear range. However, if the PageList array was constructed by some other means, or is a concatenation of two or more PageList arrays returned from various other DevHlp services, the selector mapping can be discontinuous. Because linear addresses must be mapped to physical addresses on a page-granular basis, if the PageList contains physical addresses and sizes, which do not directly correspond to page boundaries, then the selector mapping will necessarily contain *holes* that map unrequested front, or tail ends of pages that contain requested addresses. For example, given the following two PageList entries (all numbers are in hex):

- pl_PhysAddr = 4100, pl_cl = 600
- pl_PhysAddr = 1500, pl_cb = 200

PageListToGDTSelector maps the 600 hex bytes in PageList entry 1, beginning at offset 0 within the selector. However, because the underlying hardware only supports mapping integral pages, the selector also addresses the remaining 900 hex bytes in the physical page, beginning at address 4000 (4100H + 600H + 900H = 5000H = the end of the physical page). The next addresses mapped by the selector corresponds to the beginning of the first physical page described in PageList entry 2. The data for PageList entry number 2 then follows. The offsets within the selector are mapped as follows:

- 0 – 5FFH: Data for PageList entry 1.
- 600H – EFFH: Unrequested remainder of the physical page of entry 1.
- F00H – 13FFH: Unrequested beginning of the physical page of entry 2.
- 1400H – 15FFH: Data for PageList entry 2.

The first byte mapped by the selector corresponds to the first byte described in the first entry in the PageList array. The next n bytes, where n is the size parameter of the first PageList entry, is mapped contiguously from that point.

The offset within the selector of subsequent PageList entries can be computed by the formula $0 + PS - (A \bmod PS) + (B \bmod PS)$, where 0 is the offset within the selector of the byte following the end of the previous PageList entry; PS is the page size (4KB); A is the physical address of the byte following the end of the previous PageList entry; and B is the physical address of the start of the next PageList entry.

After this call has been issued for a particular selector, the addressability remains valid until the physical device driver changes its content with a subsequent call to the PageListToGDTSelector, PhysToGDTSel, PhysToGDTSelector, or LinToGDTSelector DevHlp services.

PageListToLin

This service maps physical memory pages described in an array of PageList_s structures to a linear address. The size of the linear mapping must not exceed 64KB.

Calling Sequence

```
MOV  ECX,cbSize           ; Count of bytes to be mapped;
                          ;   sum of the pl_cb fields in the PageList array
MOV  EDI,OFFSET pPageList ; Flat pointer to array of PageList_s
                          ;   structures.
MOV  DL,DevHlp_PageListToLin

CALL [Device_Help]
```

Each PageList_s structure describes a single physically contiguous subregion of the physical memory to be mapped. The format of the PageList_s structure is:

```
PageList_s struc
    pl_PhysAddr DD ? ; Physical address of first byte
                  ;   in this sub-region
    pl_cb      DD ? ; Number of contiguous bytes
                  ;   starting at pl_PhysAddr
PageList_s ends
```

Results

'C' Clear if region could be mapped.
EAX = Linear address.

'C' Set if region could not be mapped.
EAX = Error code.

Remarks: The physical memory that is being mapped must be fixed or locked prior to this call. After this call, the first byte within the returned linear range corresponds to the first byte in the first entry in the array pointed to by pPageList. If the PageList is an unmodified return array from VMLock or LinToPageList, then the mapping returned from this function is, byte for byte, the same as the original linear range. However, if the PageList array was constructed by some other means, or is a concatenation of two or more PageList arrays returned from other DevHlp services, the linear mapping might be discontinuous. Because linear addresses can only be mapped to physical addresses on a page-granular basis, if the PageList contains physical addresses and sizes, which do not directly correspond to page boundaries, then the linear mapping will necessarily contain *holes*, which map unrequested front or tail ends of pages that contain requested addresses. For example, given the following two PageList entries (all numbers are in hex):

- pl_PhysAddr = 4100, pl_cl = 600
- pl_PhysAddr = 1500, pl_cb = 200

PageListToLin maps the 600 hex bytes in PageList entry 1 to the start of the linear range. However, because the underlying hardware only supports mapping integral pages, the range also addresses the remaining 900 hex bytes in the physical page beginning at address 4000 (4100H + 600H + 900H = 5000H = the end of the physical page). The next addresses mapped by the linear range will correspond to the beginning of the first physical page described in PageList entry 2. The data for PageList entry number 2 then follows. The offsets from the starting linear address are mapped as follows:

```
100H — 6FFH: Data for PageList entry 1.
700H — FFFH: Unrequested remainder of the physical page of entry 1.
1000H — 14FFH: Unrequested beginning of the physical page of entry 2.
1500H — 16FFH: Data for PageList entry 2.
```

The first byte in the linear mapping corresponds to the first byte described in the first entry in the PageList array. The next n bytes, where n is the size parameter of the first PageList entry, are mapped contiguously from that point.

The starting linear address of subsequent PageList entries can be computed by rounding up the linear address of the end of the previous entry to a page boundary, and then adding on the low-order 12 bits of the physical address of the target PageList entry.

The linear mapping produced by this function is only valid until the caller yields the CPU, or until it issues another call to PageListToLin or PhysToVirt. PageListToLin also invalidates any outstanding PhysToVirt mappings.

PhysToGDTSel

This service maps a given GDT selector to a specified physical address, setting the access byte of the descriptor to the desired privilege value. The specified segment size must be less than or equal to 64KB.

Calling Sequence

```
MOV  EAX,PhysAddress    ; 32-bit physical address that the
                        ; GDT selector is to be mapped to
MOV  ECX,Size           ; Size of segment mapped;
                        ; must be less than or equal to 64KB
                        ; (a 0 value will map a 64KB segment)
MOV  SI,Selector        ; GDT selector (allocated using AllocateGDTSelector)
MOV  DH,Access          ; Descriptor's type and privilege level
                        ; = 0 The selector is mapped as Ring 3, readable code with
                        ; 16-bit addressing/operand size (286 format).
                        ; = 1 (binary 0001) The selector is mapped as Ring 3,
                        ; writable data.
                        ; = 3 (binary 0011) The selector is mapped as Ring 2, IOPL,
                        ; readable code with 16-bit addressing/operand size.
                        ; = 4 (binary 0100) The selector is mapped as Ring 2, IOPL,
                        ; writable data.
                        ; = 5 (binary 0101) The selector is mapped as Ring 0,
                        ; readable code with 16-bit addressing/operand size.
                        ; = 6 (binary 0110) The selector is mapped as Ring 0,
                        ; writable data.
                        ; = 128 (binary 10000000) The selector will be mapped as Ring 3,
                        ; readable code with 32-bit addressing/operand size.
                        ; = 131 (binary 10000011) The selector will be mapped as Ring 2,
                        ; IOPL, readable code with 32-bit addressing/operand size.
                        ; = 133 (binary 10000101) The selector will be mapped as Ring 0,
                        ; readable code with 32-bit addressing/operand size.
                        ; All other values are invalid.

MOV  DL,DevHlp_PhysToGDTSel

CALL [Device_Help]
```

Results

```
'C' Clear if pages locked.
    AX = Selector with the modified RPL (Requested Privilege Level) bits.

'C' Set if segment unavailable.
    EAX = Error code.
        Possible errors:
            Invalid address
            Invalid selector
            Invalid parameter
```

Remarks: The physical memory that is being mapped must be fixed or locked prior to the call to this service. After the call has been issued for a particular selector, the addressability remains valid until the physical device driver changes its content with a subsequent call to the DevHlp services, PhysToGDTSel, PhysToGDTSelector, PageListToGDTSelector, or LinToGDTSelector.

PhysToGDTSelector

This service converts a 32-bit address to a Global Descriptor Table (GDT) selector-offset pair.

Calling Sequence

```
MOV  AX,address_high      ; 32-bit physical address
MOV  BX,address_low       ;
MOV  CX,length            ; Length of segment
MOV  SI,selector          ; Selector to be set up
MOV  DL,DevHlp_PhysToGDTSelector
```

```
CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

Invalid address

Invalid selector

Remarks: This function is used to provide addressability through a GDT selector to data. The addressability of the GDT selector remains valid and unchanged until another call to PhysToGDTSelector is made for the same selector.

The DevHlp, AllocGDTSelector, is used at initialization (INIT) time to allocate the GDT selectors that the physical device driver can use with PhysToGDTSelector.

PhysToGDTSelector creates selector:offset addressability for a 32-bit physical address. The selector created, however, does not represent a normal memory segment, such as those usually managed by the operating system, and is more of a *fabricated segment* for private use by the physical device driver. Such a segment cannot be passed on system calls, and can be used by the physical device driver only to fetch data.

PhysToUVirt

This service converts a 32-bit physical address to a valid selector:offset pair addressable out of the current Local Descriptor Table (LDT). Additional information about the selector can be retained by the memory manager, if special processing, based on the tag type, is required.

Calling Sequence

```
MOV  AX,address_high ; 32-bit physical address (or selector, if request type 2)
MOV  BX,address_low  ;
MOV  CX,length        ; Length of area (less than or equal to 65535, 0 = 65536)
MOV  DH,request_type  ; Type of Request
                        ; 0 = get virtual address, make segment readable/executable,
                        ;    with Application Program Privilege
                        ; 1 = get virtual address, make segment readable/writable,
                        ;    with Application Program Privilege
                        ; 2 = free virtual address (OS/2 mode only)
                        ; 3 = get virtual address, make segment readable/executable,
                        ;    with I/O Privilege
                        ; 4 = get virtual address, make segment readable/writable,
                        ;    with I/O Privilege
                        ; 5 = get virtual address, make segment readable/writable,
                        ;    with Application Program Privilege and assign it with a tag
MOV  SI,tag_type      ; Type of tag (used only on request type 5, preserved on all others)
                        ; 0 = foreground only video selector
MOV  DL,DevHlp_PhysToUVirt
CALL [Device_Help]
```

Results

'C' Clear if successful.
 ES:BX = segment/selector:offset pair
 (for request types 0, 1, 3, and 4).

'C' Set if error.

Remarks: This service is typically used to provide a caller of a physical device driver with addressability to a fixed memory area, like ROM code and data. The physical device driver must know the physical address of the memory area to be addressed. PhysToUVirt leaves its result in ES:BX. This service also can be used to free a selector returned on a prior call to PhysToUVirt. For request type 2, AX contains a selector on entry to PhysToUVirt. BX and CX are ignored.

Note: If the input physical address is not on a 4KB boundary, then the length of the result segment will be limited to 64KB minus the offset from 4KB of the input physical address.

PhysToUVirt creates selector:offset LDT addressability for a 32-bit physical address. This function is provided so a physical device driver can give an application process addressability to a fixed memory area, such as in the BIOS-reserved range from 640KB to 1MB. It can also be used to give a client application addressability to a data segment of the physical device driver. The selector created, however, does not represent a normal memory segment such as those usually managed by the operating system, and is more of a *fabricated segment* for private use between a physical device driver and an application. Data within such a segment cannot be passed on system calls, and can be used by the receiving application only to return data variables.

PhysToVirt

This service converts a 32-bit address to a valid selector:offset pair.

Calling Sequence

```
MOV  AX,address_high      ; 32-bit physical address
MOV  BX,address_low       ;
MOV  CX,length            ; Length of segment
MOV  DH,result            ; Leave result...
                                ; 0 = in DS:SI
                                ; 1 = in ES:DI

MOV  DL,DevHlp_PhysToVirt

CALL [Device_Help]
```

Results

```
'C' Clear if successful.
    If DH was set to 0 on input:
        DS:SI = Converted virtual address.
        ES    No mode switch, ES is preserved.
                Mode switch, if ES contains the address of the
                device driver data segment on input, it will
                be converted to a valid virtual address.
                Otherwise, it is set to zero.
    If DH was set to 1 on input:
        ES:DI = Converted virtual address.
        DS    No mode switch, DS is preserved.
                Mode switch, if DS contains the address of the
                device driver data segment on input, it will
                be converted to a valid virtual address.
                Otherwise, it is set to zero.

'C' Set if error.
    AX = Error code.

'Z' Clear if no change in addressing mode.

'Z' Set if addressing mode has changed;
    (previously stored addresses must be recalculated).
```

Remarks: This function leaves its result in ES:DI or DS:SI, giving the physical device driver the ability to move strings in either direction. The returned virtual address does not remain valid, if the device driver blocks or yields control. The returned virtual address can also be destroyed, if the physical device driver routine that issues the DevHlp, PhysToVirt, calls another routine.

While pointers generated by this routine are in use, the device driver can call only another PhysToVirt request. No other DevHlp routines can be called, because they might not preserve the special DS or ES values created by the call to PhysToVirt. The performance characteristics of PhysToVirt are highly variable.

PhysToVirt preserves the registers CS, SS, SP, and DS, if called with DH=1, or the ES register, if called with DH=0. The pool of temporary selectors used by PhysToVirt in the OS/2 mode is not dynamically extendable. The converted addresses are valid as long as the physical device driver does not relinquish control (Block, Yield, or RET). An interrupt handler can use converted addresses prior to its EOI, with interrupts enabled. If an interrupt handler needs to use converted addresses after its EOI, it must protect the converted addresses by running with interrupts disabled.

The virtual mapping produced by the DevHlp, PhysToVirt, is invalidated by calls to the DevHlp services, PageListToLin and PhysToVirt.

The task-time strategy routine of a physical device driver can run enabled on a PS/2 system.

The Segment Length parameter must be set to the length of the transfer.

Using DevHlp Services for Address Conversion: There are some basic guidelines when using the DevHlp service, PhysToVirt, for address conversion.

- Use ES:DI whenever possible when converting a single physical address
- Use ES:DI for the first address conversion when using two physical addresses
- Check the physical address pair, and convert the physical address above 1MB first.

The following examples are recommended ways of using these DevHlp's in various situations. These examples apply to both task-time and interrupt-time operations, except where noted:

- To convert a single physical address to use as the source in a data transfer to a logical address, (that is, one that was passed as input for this data transfer request):
 1. Save DS
 2. Call PhysToVirt with DS:SI for the converted address
 3. Perform the data transfer
 4. Restore DS
- To provide two logical addresses in order to do a data transfer:
 1. Examine the physical address pair. If one of the physical addresses is above 1MB, convert it first.
 2. Call PhysToVirt with ES:DI for the first address.
 3. Save DS.
 4. Call PhysToVirt with DS:SI for the second address.
 5. Perform the data transfer.
 6. Restore DS.
- To do multiple data transfers:
 1. Examine the first physical address pair. If one of the physical addresses is above 1MB, convert it first.
 2. Call PhysToVirt with ES:DI for the first address.
 3. Save DS.
 4. Call PhysToVirt with DS:SI for the second address.
 5. Perform the data transfer.
 6. Restore DS.
 7. Examine the second physical address pair. If one of the physical addresses is above 1MB, convert it first.
 8. Call PhysToVirt with ES:DI for the first address.
 9. Save DS.
 10. Call PhysToVirt with DS:SI for the second address.
 11. Perform the data transfer.
 12. Restore DS.
 13. Perform these steps until all data transfers are complete.
- To provide two logical addresses for a data transfer, which must be broken down into smaller chunks in order to *yield* periodically:
 1. Examine the physical address pair. If one of the physical addresses is above 1MB, convert it first.
 2. Call PhysToVirt with ES:DI for the first address.
 3. Save DS.
 4. Call PhysToVirt with DS:SI for the second address.
 5. Perform the data transfer on the chunk.
 6. Restore DS.

7. YIELD.
 8. When control is returned, repeat these steps.
- To pass two logical addresses to a subroutine, one of which must be converted from a physical address. The other is obtained from the device driver's data segment:
 1. Examine the physical address pair. If one of the physical addresses is above 1MB, convert it first.
 2. Call PhysToVirt with ES:DI for the address to convert.
 3. Save the converted address in the appropriate input parameter to the subroutine.
 4. Save the other logical address (located in the device driver's data segment) in the appropriate input parameter to the subroutine.
 5. Call the subroutine.
 - To use two logical addresses to do a data transfer at interrupt time before the EOI:
 1. Examine the physical address pair. If one of the physical addresses is above 1MB, convert it first.
 2. Call PhysToVirt with ES:DI for the first address.
 3. Save DS.
 4. Call PhysToVirt with DS:SI for the second address.
 5. Perform the data transfer.
 6. Restore DS.
 7. Issue the EOI.
 - To use two logical addresses in order to do a data transfer at interrupt time after the EOI:
 1. Issue the EOI.
 2. Examine the physical address pair. If one of the physical addresses is above 1MB, convert it first.
 3. Save the interrupt flag.
 4. Disable interrupts.
 5. Call PhysToVirt with ES:DI for the first address.
 6. Save DS.
 7. Call PhysToVirt with DS:SI for the second address.
 8. Perform the data transfer.
 9. Restore DS.
 10. Restore the interrupt flag.

PostEventSem

This service posts an event semaphore that was previously reset with ResetEventSem. If the event is already posted, the post count is incremented and the ERROR_ALREADY_POSTED return code is returned. Otherwise, the event is posted, the post count is set to one, and all threads that called DosWaitEventSem are scheduled to run.

Calling Sequence

```
MOV  EAX, SemaphoreHandle    ; DWORD semaphore handle
MOV  DL, DevHlp_PostEventSem

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

EAX = Error code.

Possible errors:

Already posted

Invalid handle

Too many posts

Remarks: This function uses the EAX and Flags registers. PostEventSem can be called only from a Ring 0 device driver or file system driver. The handle passed in must be a handle to a *shared event* semaphore. If the handle does not exist, or is not a shared event semaphore, or if the semaphore was not previously opened with OpenEventSem, then ERROR_INVALID_HANDLE is returned. There is a limit of 65,534 (64KB – 1) posts allowed per event semaphore. If this limit is reached, then the ERROR_TOO_MANY_POSTS return code is returned.

To reverse this operation, call ResetEventSem. This resets the event, so that any threads that subsequently wait on the event semaphore (with DosWaitEventSem) will be blocked.

ProtToReal

This service allows a physical device driver to change the processor mode from protect mode to real mode at interrupt time.

Remarks: This function still exists for compatibility reasons, but is no longer operational.

PullParticular

This service pulls the specified packet from the selected request packet linked list. If the packet is not found, then an indicator is set on return.

Calling Sequence

```
MOV  SI,OFFSET DS:queue      ; Location of queue head (should match PushRequest value)
LES  BX,request_packet       ; Pointer to request packet.
MOV  DL,DevHlp_PullParticular

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if the specified request is not found.

Remarks: A physical device driver uses PushReqPacket and PullReqPacket to maintain a work queue for each of its devices. PullParticular is used to remove a specific request packet from the work queue, typically when a process has terminated before finishing its I/O. PullParticular can also be used to remove request packets that were allocated by AllocReqPacket from the request packet linked list.

PullReqPacket

This service pulls the next waiting request packet from the selected request packet linked list. If there is no packet in the list, then an indicator is set on return.

Calling Sequence

```
MOV  SI,OFFSET DS:queue      ; Location of queue head (should match PushReqPacket value)
MOV  DL,DevHlp_PullReqPacket

CALL [Device_Help]
```

Results

'C' Clear if successful.
ES:BX = Pointer to the request packet.

'C' Set if there is no request packet in the list.

Remarks: A physical device driver uses PushReqPacket and PullReqPacket to maintain a work queue for each of its devices and units. The physical device driver must provide the storage for the DWORD work queue head that defines the start of the request packet linked list. The work queue head must be initialized to 0. PullReqPacket can also be used to remove request packets that were allocated by AllocReqPacket from the request packet queue.

PushReqPacket

This service adds the current device request packet to the linked list of packets to be executed by the physical device driver.

Calling Sequence

```
MOV  SI,OFFSET DS:queue      ; Location of the DWORD queue head
                                ; (which points to the first request in the list)
LES  BX,request_packet        ; Pointer to request packet.

MOV  DL,DevHlp_PushReqPacket

CALL [Device_Help]
```

Results: None.

Remarks: A physical device driver uses PushReqPacket and PullReqPacket to maintain a work queue for each of its devices. The physical device driver must provide the storage for the DWORD work queue head, which defines the start of the request packet linked list. The work queue head must be initialized to 0.

The physical device driver task-time thread adds all incoming Read/Write requests to its Request List. The device driver task-time thread then determines whether the interrupt-time thread is active, and if not, it sends the request to the device. Because the device can be active at this point, the device driver task-time thread must turn off interrupts before calling the device; otherwise, a window exists in which the device finishes before the packet is put on the list.

PushReqPacket can also be used to place request packets that were allocated by AllocReqPacket in the request packet work queue.

QueueFlush

This service clears the character queue structure that is specified; that is, it empties the buffer.

Calling Sequence

```
MOV  BX,OFFSET DS:queue      ; Points to the queue structure to be flushed.  
                                ; (The Qsize field must be set up.)  
MOV  DL,DevHlp_QueueFlush  
CALL [Device_Help]
```

Results: None.

Remarks: This function operates on the simple character queue structure initialized by QueueInit.

QueueInit

This service initializes the specified character queue structure.

Calling Sequence

```
MOV  BX,OFFSET DS:queue      ; Points to the queue structure to be initialized.  
                                ; (The Qsize field must be set up.)  
MOV  DL,DevHlp_QueueInit  
CALL [Device_Help]
```

Results: None.

Remarks: This function must be called before any other queue manipulation subroutine. Prior to this call, the physical device driver must allocate the character queue buffer with the following queue header:

Qsize	DW	?	;size of queue in bytes
Qchrout	DW	?	;index of next char out
Qcount	DW	?	;count of chars in the queue
Qbase	DB	?	;start of queue buffer

The *Qsize* field should be initialized by the physical device driver before calling QueueInit.

QueueRead

This service returns and removes a character from the beginning of the specified character queue structure. If the queue is empty, an indicator is set.

Calling Sequence

```
MOV  BX,OFFSET DS:queue      ; Points to the queue structure.
MOV  DL,DevHlp_QueueRead

CALL [Device_Help]
```

Results

'C' Clear if successful.
AL = the character read from the queue.

'C' Set if the queue is empty.

Remarks: This function operates on the simple character queue structure initialized by QueueInit.

QueueWrite

This service adds a character at the end of the specified character queue structure. If the queue is full, an indicator is set.

Calling Sequence

```
MOV  BX,OFFSET DS:queue    ; Points to the queue structure.
MOV  AL,char               ; Character to insert at the end of the queue.
MOV  DL,DevHlp_QueueWrite

CALL [Device_Help]
```

Results

'C' Clear if character stored successfully.

'C' Set if queue is full.

Remarks: This function operates on the simple character queue structure initialized by QueueInit.

RealToProt

This service allows a physical device driver to change the processor mode from real mode to protect mode at interrupt time.

Remarks: This function still exists for compatibility reasons but is no longer operational.

Register

This service adds a monitor to the chain of monitors for a class of device.

Calling Sequence

```
LES  SI,input_buffer      ; Address of input buffer
MOV  DI,output_buffer_offset ; Offset of output buffer
MOV  CX,monitor_PID       ; Process ID of monitor task
MOV  AX,monitor_handle     ; Handle for chain returned from previous MonitorCreate call
MOV  DH,placement_flag     ; High or Low place in chain
MOV  DL,DevHlp_Register
```

```
CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

Invalid monitor handle

Invalid parameter

Not enough memory

Remarks: This function can be called only at task time in the OS/2 mode. If this function is called in a DOS Session context, an invalid parameter error is returned to the physical device driver.

A monitor chain must have previously been created with MonitorCreate. A single process can register more than one monitor (with different input and output buffers) with the same monitor chain. The first WORD of each of the input and output buffers must contain the length in bytes (length WORD inclusive) of the buffers. The length of the input and output buffers of the monitor must be greater than the length of the monitor chain buffer of the physical device driver plus 20 bytes. The input buffer, output buffer offset, and placement flag are supplied to the physical device driver by the monitor application that is requesting monitor registration (that is, calling DosMonReg).

The physical device driver must identify the monitor chain with the monitor handle returned from a previous MonitorCreate call. The physical device driver can determine the Process ID of the requesting monitor task from the Local InfoSeg. See "GetDOSVar" on page 17-36.

RegisterBeep

This service is called by the physical Clock\$ device driver during initialization time to register the beep service entry point so that other physical device drivers or the kernel can generate preemptive beeps.

Calling Sequence

```
MOV  CX,SEGMENT PDDBeep_Func    ; CX:DI (16:16) = Entry point to
MOV  DI,OFFSET PDDBeep_Func    ; physical device driver's Beep routine.
MOV  DL,DevHlp_RegisterBeep
```

```
CALL [Device_Help]
```

Results

Success:

 If successful, AX = 0.

Failure:

 N/A

RegisterPDD

This service registers a 16:16 physical device driver for physical device driver/virtual device driver (PDD/VDD) communication with a virtual device driver. The function is used by a physical device driver to register its name and a communication entry point with the DOS Session Manager. Later, a virtual device driver can use VDHOpenPDD to open communication with the physical device driver.

Calling Sequence

```
MOV    DS,SEGMENT PDD_Name      ; DS:SI = Pointer to ASCIIZ name for
                                ; the physical device driver
LEA     SI,PDD_Name
MOV     DI,SEGMENT PDD_Function
MOV     ES,DI                   ; ES:DI = Pointer to physical device driver's
                                ; communication function
LEA     DI,PDD_Function
MOV     DL,DevHlp_RegisterPDD

CALL    [Device_Help]
```

Results

'C' Clear if successful.
AX = 0.

If the function fails, a system halt will occur.

Remarks: If ES:DI is NULL (0:0), this call removes the registration of the name of this physical device driver. The physical device driver name supplied to this service does not need to match the string in the physical device driver's header.

If a physical device driver deactivates itself, it must close down any interaction with virtual device drivers. If a physical device driver registers an entry point during initialization, but fails later during initialization, it must call this function with a NULL function pointer in order to remove the registration.

RegisterStackUsage

This service indicates the expected stack usage of the physical device driver to the interrupt manager.

Calling Sequence

```
MOV  BX,OFFSET DS:StackUsage
MOV  DL,DevHlp_RegisterStackUsage

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if stack usage exceeds system maximum.
If this happens, the physical device driver must deinstall itself.

Remarks: The StackUsage data structure has the following format:

```
StackUsage  STRUC
SU_cbStruct  DW 14      ; Number of bytes in structure, including itself.
SU_flags     DW ?       ; Bit 0x0001 'on' indicates that the interrupt
                        ; procedure enables interrupts. All other bits are reserved.
SU_iIRQ      DW ?       ; IRQ of interrupt handler that is being
                        ; described by the following data.
SU_cbStackCLI DW ?       ; Number of bytes of stack used in the
                        ; interrupt procedure when interrupts are disabled.
SU_cbStackSTI DW ?       ; Number of bytes of stack used after interrupt
                        ; procedure enables interrupts.
SU_cbStackEOI DW ?       ; Number of bytes of stack used after interrupt
                        ; procedure issues EOI.
SU_cNest     DW ?       ; Maximum number of levels that the device
                        ; driver expects to nest.
StackUsage  ENDS
```

If the physical device driver interrupt routines nest greater than the specified number, the interrupt manager disables the IRQ at the PIC, for the remainder of the boot session. A device must issue RegisterStackUsage once for each IRQ that it services.

OS/2 2.0 supports a total of 8KB of interrupt stack.

RegisterTmrDD

This service sends the physical device driver pointers to the Timer value and Timer rollover count in kernel address space.

Calling Sequence

```
MOV  CS,SEGMENT PTimer0_Entry_Point
MOV  DI,OFFSET PTimer0_Entry_Point
MOV  DL,DevHlp_RegisterTmrDD
```

```
CALL [Device_Help]
```

Results

Success:

- DI:BX = qwTmrRollover (16:16) in kernel address space.
- DI:CX = qwTmr (16:16) in kernel address space.
 - = NULL if Doshlp function not present.

Remarks: RegisterTmrDD is callable only at physical Timer device driver initialization time. This function uses the EAX, BX, CX, DI, and EFlags registers.

ResetEventSem

This service resets an event semaphore that has been previously opened with OpenEventSem. The number of *posts* performed on the event before it was reset is returned to the caller in the pulPostCt parameter. If the event was already reset, the ERROR_ALREADY_RESET return code is returned, and zero is returned in the pulPostCt parameter. It is not reset a second time.

Calling Sequence

```
MOV  EAX, SemaphoreHandle    ; DWORD semaphore handle
MOV  EDI, pNumPosts          ; Pointer to variable to receive the number of
                               ; posts performed on the event before the reset.
MOV  DL, DevHlp_ResetEventSem
CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

EAX = Error code.

Possible errors:

Already reset

Invalid handle

Remarks: This function uses the EAX, EDI, and Flags registers. ResetEventSem can be called only from a Ring 0 device driver or file system driver. The handle *passed in* must be a handle to a *shared event* semaphore. If the handle does not exist, or is not a shared event semaphore, or if the semaphore was not previously opened with OpenEventSem, then ERROR_INVALID_HANDLE is returned.

To reverse this operation, call PostEventSem. This posts the event, so that any threads that were waiting for the event semaphore to be posted (with DosWaitEventSem) are allowed to run.

ResetTimer

This service removes a timer handler for the physical device driver.

Calling Sequence

```
MOV  AX,Offset cs:timer_handler    ; Offset to timer handler
MOV  DL,DevHlp_ResetTimer

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

Invalid handler

Remarks: ResetTimer removes a timer handler from the list of timer handlers. Timer handlers are analogous to the user timer interrupt (INT 1CH). Refer to “TickCount” on page 17-90, for a discussion of timer handlers.

DS should be set to the data segment of the physical device driver. If the physical device driver issued a call to PhysToVirt referencing the DS register, the physical device driver will restore DS to the original value.

ROMCritSection

This service is used to flag a critical section of execution in a physical device driver's DOS mode software interrupt handler. This prevents the DOS mode from being suspended in the background.

Remarks: This function still exists for compatibility reasons but is no longer operational.

Run

This service is the companion routine to Block. When Run is called, it awakens all the threads that were blocked for this particular event identifier.

Calling Sequence

```
MOV  BX,event_id_low      ; Low word of event identifier.  
MOV  AX,event_id_high     ; High word of event identifier.  
MOV  DL,DevHlp_Run  
  
CALL [Device_Help]
```

Results

AX = Count of threads awakened.

'Z' Set or clear according to the contents of AX.

Remarks: Run returns immediately to its caller; the awakened threads are run at the next available opportunity. Run is often called at interrupt time. See "Block" on page 17-21 for a more detailed discussion of blocking and running threads.

SchedClockAddr

This service allows the physical Clock\$ device driver to obtain a pointer to the address of the system's clock tick handler, SchedClock. SchedClock must be called on each occurrence of a periodic clock tick.

Calling Sequence

```
MOV  AX,Pointer_Save      ; Offset in DS to a DWORD where the pointer is returned
MOV  DL,DevHlp_SchedClockAddr
CALL [Device_Help]
```

Results

Pointer_Save will contain the address of the system tick handler.

Remarks: The physical Clock\$ device driver calls this function during the physical Clock\$ device driver's initialization. For input to this DevHlp, the physical Clock\$ device driver must ensure that its DS points to its data segment, and supply the offset to a DWORD area. SchedClockAddr then fills in the physical device driver's save area with a pointer to a DWORD in system memory. The DWORD in system memory contains the pointer to the SchedClock routine. This is shown in the following example:

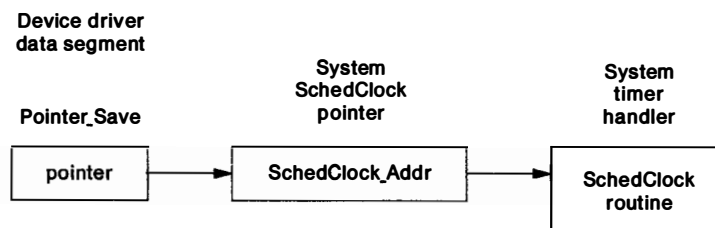


Figure 17-1. DWORD in System Memory

The physical device driver can use the pointer that is returned in Pointer_Save to call SchedClock. SchedClock is called by the physical Clock\$ device driver with a CALL FAR INDIRECT, using the pointer to the area that contains the actual address of the SchedClock routine.

SchedClock must be called at interrupt time for each periodic clock tick to indicate the passage of system time. The tick is then dispersed to the appropriate components of the system. A definition of the interface to SchedClock follows:

```
MOV  AL,millisecs        ; Milliseconds since last call
MOV  DH,EOfIflag         ; Indicator of EOI
                                ; = 0 prior to EOI
                                ; = 1 after EOI
CALL [pointer]           ; Pointer to the area which contains the actual address
                                ; of SchedClock
```

The Clock\$ device driver's interrupt handler must run with interrupts enabled as the convention, prior to issuing an End-Of-Interrupt (EOI) for the timer interrupt. Any critical processing, such as updating the fraction-of-seconds count, must be done prior to calling SchedClock. SchedClock must then be called to allow system processing prior to the dismissal of the interrupt. When SchedClock returns, the Clock\$ device driver must issue the EOI, and call SchedClock again. Notice that once the EOI has been issued, the device driver's interrupt handler can be reentered. SchedClock is also reentrant.

The physical device driver must not get the actual address of the SchedClock routine, but instead use the pointer returned by SchedClockAddr.

SemClear

This service releases a semaphore and restarts any blocked threads waiting on the semaphore.

Calling Sequence

```
MOV  BX,sem_handle_low    ; Semaphore handle
MOV  AX,sem_handle_high   ;
MOV  DL,DevHlp_SemClear

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

```
ERROR_INVALID_HANDLE
ERROR_EXCL_SEM_ALREADY_OWNED
ERROR_INVALID_AT_INTERRUPT_TIME
ERROR_PROTECTION_VIOLATION
```

Remarks: A physical device driver can clear either a RAM semaphore or a system semaphore. The physical device driver can obtain (own) a semaphore through SemRequest. The semaphore handle for a RAM semaphore is the virtual address of the doubleword of storage allocated for the semaphore.

To access a RAM semaphore at interrupt time, the physical device driver must locate the semaphore in the physical device driver's data segment (DS). For a system semaphore, the handle must be passed to the physical device driver by the caller by way of a generic IOCTL. The device driver must convert the caller's handle to a system handle with SemHandle.

A RAM semaphore can be cleared at interrupt time only if it is in storage that is directly addressable by the physical device driver; that is, in the physical device driver's data segment.

SemHandle

This service provides a semaphore handle to the physical device driver.

Calling Sequence

```
MOV  BX,sem_key_low    ; Semaphore identifier
MOV  AX,sem_key_high   ;
MOV  DH,usage_flag     ; Indicates if in use
                        ; 0 = Not-In-Use
                        ; 1 = In-Use
MOV  DL,DevHlp_SemHandle

CALL [Device_Help]
```

Results

```
'C' Clear if successful.
    AX:BX = The system handle.

'C' Set if error.
    AX = Error code.
    Possible errors:
        Invalid handle for the semaphore if DH = 1
```

Remarks: SemHandle is used to convert the semaphore handle (or user key), provided by the caller of the physical device driver, to a system handle that the physical device driver can use. This handle then becomes the key that the physical device driver uses to reference the system semaphore. This allows the system semaphore to be referenced at interrupt time by the physical device driver. This key is also used when the device driver is finished with the system semaphore. The physical device driver must call SemHandle with the usage_flag indicating that the device driver is finished with the system semaphore.

SemHandle is called at task time to indicate that the system semaphore is In-Use, and is called at either task time or interrupt time to indicate that the system semaphore is Not-In-Use. *In-Use* means that the physical device driver might be referencing the system semaphore. *Not-In-Use* means that the physical device driver has finished using the system semaphore and will not reference it again.

The key of a RAM semaphore is its virtual address, where virtual address is the generic term for selector:offset. SemHandle can be used for RAM semaphores. Because RAM semaphores have no system handles, SemHandle simply returns the RAM semaphore key back to the caller. A physical device driver can determine that a semaphore is a RAM semaphore, if the key remains unchanged after returning from the SemHandle function. If the key returned from SemHandle is different from the one passed to the function, then the physical device driver can determine that it is a handle for a system semaphore.

If C (the carry flag) is set on return from this function, the physical device driver should call the DevHlp, VerifyAccess, with TypeOfAccess set to Read/Write, before assuming this is a RAM semaphore. If a RAM semaphore is to be used, it must be accessed only at task time unless it is in locked storage.

It is necessary to call SemHandle at task time to indicate that a system semaphore is In-Use because:

- The caller-supplied semaphore handle refers to task-specific system semaphore structures. These structures are not available at interrupt time, so SemHandle converts the task-specific handle to a system-specific handle. For uniformity, the other semaphore DevHlp functions accept only system-specific handles regardless of the mode (task time or interrupt time).
- An application could delete a system semaphore while the physical device driver is using it. If a second application were to create a system semaphore soon after, the system structure used by the original semaphore could be reassigned. A physical device driver that tries to manipulate the semaphore of the original process would inadvertently manipulate the semaphore of the new process. Therefore, the

DevHlp_SemHandle

SemHandle In-Use indicator increases a counter so that, although the calling thread can still delete its task-specific reference to the semaphore, the semaphore remains in the system structures.

A physical device driver must subsequently call SemHandle with Not-In-Use when use of the semaphore is complete, so that the system semaphore structure can be freed. There must be a call to indicate Not-In-Use to match every call to indicate In-Use (1:1 relationship).

The state of the interrupt flag is not preserved across calls to this DevHlp.

SemRequest

This service claims a semaphore. If the semaphore is already owned, the thread in the physical device driver is blocked until the semaphore is released or until a timeout occurs.

Calling Sequence

```
MOV  BX,sem_handle_low  ; Semaphore handle
MOV  AX,sem_handle_high ;
MOV  CX,sem_timeout_low ; Timeout value in milliseconds
MOV  DI,sem_timeout_high ;
                        ; -1 = wait forever
                        ; 0 = no wait if semaphore owned
                        ; > 0 = timeout

MOV  DL,DevHlp_SemRequest

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

```
ERROR_SEM_TIMEOUT
ERROR_SEM_OWNER_DIED
ERROR_INVALID_HANDLE
ERROR_TOO_MANY_SEM_REQUESTS
ERROR_INTERRUPT
ERROR_PROTECTION_VIOLATION
```

Remarks: This function checks the state of the semaphore. If it is unowned, SemRequest marks it *owned* and returns immediately to the caller. If the semaphore is owned, SemRequest blocks the device driver thread until the semaphore is unowned, then tries again. The timeout parameter is used to place an upper limit on the amount of time to block before returning to the requesting device driver thread. "SemClear" on page 17-80 is used at either task time or interrupt time to release the semaphore.

For a system semaphore, the handle must be passed to the device driver by the caller through a generic IOCTL. The physical device driver must convert the caller's handle to a system handle with SemHandle. Note that this service is valid in user mode only for RAM semaphores. System semaphores are not available in user mode by device drivers.

The state of the interrupt flag is not preserved across calls to this DevHlp.

SendEvent

This service is called by a physical device driver to indicate the occurrence of an event.

Calling Sequence

```
MOV  AH,event          ; Event being signalled
MOV  BX,argument       ; Parameter for the event being signalled
MOV  DL,DevHlp_SendEvent

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error sending signal.

Remarks: The events are defined in the following manner:

- Session manager hot key from the mouse:

Event = 0. Reserved.

Argument = 2-byte time stamp, where the high-order byte is seconds, and the low-order byte is hundredths of seconds.

- Ctrl + Break:

Event = 1

Argument = 0. Reserved.

- Ctrl + C:

Event = 2

Argument = 0. Reserved.

- Ctrl + NumLock:

Event = 3

Argument = Foreground session number.

- Ctrl + PrtSc:

Event = 4

Argument = 0. Reserved.

- Shift + PrtSc:

Event = 5

Argument = 0. Reserved.

- Session Manager hot key from the keyboard:

Event = 6

Argument = Hot Key ID. The physical Keyboard device driver uses the Hot Key ID, which is set through the use of the IOCTL "Function 56H – Set Session Manager Hot Key" on page 18-81.

- Reboot key sequence from the keyboard:

Event = 7

Argument = 0. Reserved.

SetIRQ

This service to set a hardware interrupt vector to the physical device driver interrupt handler.

Calling Sequence

```
MOV  AX,Offset CS:handler ; Interrupt handler offset
MOV  BX,IRQnum           ; Interrupt level number (0 - FH)
MOV  DH,Shared_Int       ; Interrupt sharing (0 = not shared, 1 = shared)
MOV  DL,DevHlp_SetIRQ
```

```
CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

IRQ is not available

Remarks: The attempt to register an interrupt handler for an IRQ to be shared will fail under any of the following circumstances:

- If the IRQ is already owned by another device driver as *not-shared*
- If the IRQ is the IRQ used to cascade the slave 8259 interrupt controller (IRQ 2).

The attempt to register an interrupt handler for an IRQ to be *not-shared* will fail under any of the following circumstances:

- If the IRQ is already owned by another device driver as *shared* or *not-shared*
- If the IRQ is the IRQ used to cascade the slave 8259 interrupt controller.

DS should be set to the physical device driver's data segment. If the physical device driver has issued a call to PhysToVirt referencing the DS register, it should restore DS to its original value.

The IRQnum value is range checked and C (the carry flag) is set, if IRQnum is not in the range of 0–0FH.

Hardware interrupt sharing is not supported on all systems. A SetIRQ request to share an interrupt level on a system where sharing is not supported returns an error. See "Hardware Interrupt Management" on page 4-4 for a discussion of the limitations on hardware interrupt sharing and the systems supported.

SetROMVector

This service is used to replace a DOS-mode software interrupt handler with a handler from the physical device driver. This service returns a DOS-mode pointer to the previous software interrupt handler for chaining.

Remarks: This function still exists for compatibility reasons but is no longer operational.

SetTimer

This service adds a timer handler to the list of timer handlers to be called on a timer tick.

Calling Sequence

```
MOV  AX,OFFSET CS:timer_handler    ; Offset of timer handler.
MOV  DL,DevHlp_SetTimer

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

Timer handler disallowed

(the maximum number of handlers has been reached
or the timer handler is already set)

Remarks: This function is a subset of the DevHlp, TickCount. SetTimer allows a physical device driver to add a timer handler to a list of timer handlers called on every timer tick. A physical device driver can use a timer handler to drive a non-interrupt device instead of using timeouts with the Block and Run services. Block and Run are costly on a character-by-character basis; the cost is one or more task switches for each character I/O. Timer handlers are required to save and restore registers.

A maximum of 32 timer handlers are available in the system. While a timer handler is in the format of a FAR CALL/RETURN routine, it operates in interrupt state. The timer handler is analogous to the user timer (INT 1CH) handler. Care should be taken not to remain in the handler very long. Timer handlers are responsible for saving and restoring registers on entry to and exit from this function.

DS should be set to the data segment of the physical device driver. If the physical device driver has issued a call to PhysToVirt referencing the DS register, it will restore DS to the original value.

SortReqPacket

This service is used by block (Disk) device drivers to add a new request to their work queue. This routine inserts the request packet in the linked list of request packets in the order of starting sector number.

Calling Sequence

```
MOV    SI,OFFSET DS:queue      ; Location to DWORD queue head (which points to
                                ; the first request). It should be initialized to 0.
LES    BX,request_packet       ; Pointer to request packet.
MOV    DL,DevHlp_SortReqPacket

CALL   [Device_Help]
```

Results: None.

Remarks: The sorting by sector number is designed to reduce the length and number of disk head seeks. This is a simple algorithm, and does not account for multiple heads that operate on the media, or for target drive in the request packet. SortReqPacket inserts the current request packet into the specified linked list of packets, sorted by starting sector number. SortReqPacket can be used to place request packets that were allocated by AllocReqPacket in the request packet queue.

TCYield

This service is similar to the Yield DevHlp service, except that TCYield yields the CPU only to a time-critical thread if one is available.

Calling Sequence

```
MOV    DL,DevHlp_TCYield  
CALL   [Device_Help]
```

Results: None.

Remarks: This function is a subset of the Yield function; it is not necessary for the physical device driver to do both a Yield and a TCYield. Physical device drivers, particularly those that perform program I/O on long strings of data or that poll a device, are the one part of the kernel that can use a large amount of time in the CPU. These physical device drivers should periodically check the TCYield flag and call the TCYield function to yield the CPU to a time-critical thread. The location of the TCYield flag is obtained from a call to GetDOSVar. For performance reasons, the physical device driver checks the TCYield Flag once every 3 milliseconds. If the flag is set, then the physical device driver calls TCYield. Because the physical device driver can relinquish control of the CPU, it should not assume that the state of the interrupt flag is preserved across a call to TCYield.

TickCount

This service registers a new timer handler or modifies a previously registered timer handler to be called on every n timer ticks instead of every timer tick.

Calling Sequence

```
MOV  AX,OFFSET CS:timer_handler ; Offset to timer handler
MOV  BX,count                   ; Value of  $n$ ; the number of tick counts
                                   ; (0 - FFFF)
                                   ; 0 means FFFFH+1 ticks
MOV  DL,DevHlp_TickCount
CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

AX = Error code.

Possible errors:

Timer handler cannot be modified or set

Remarks: A physical device driver can use a timer handler to drive a non-interrupt device instead of using timeouts with the Block and Run services. Block and Run are costly on a character-by-character basis; the cost is one or more task switches for each character I/O. Timer handlers are required to save and restore registers. While a timer handler is in the format of a FAR CALL/RETURN routine, it operates at interrupt time. The timer handler is analogous to the user timer (INT 1CH) handler. As with any handler that runs in an interrupt context, care must be taken to make the execution path as short as possible.

For a new timer handler, TickCount registers the timer handler to be called on every n timer ticks instead of every timer tick. For a previously registered timer handler, TickCount changes the number of ticks that must take place before the timer handler gets control. This allows physical device drivers to support the timeout function without needing to count ticks.

At task time, this DevHlp can be used to modify a timer handler registered through SetTimer, or to register a new timer handler that is initially invoked every n ticks. In user mode (task time), TickCount can only be used to modify a timer handler already registered. In interrupt mode (interrupt time), this service can only be used to modify a timer handler already registered. This allows an interrupt handler to reset the timing condition at interrupt time.

Notice that SetTimer sets a default of n ticks to 1. Multiple TickCount requests can be issued for a given timer handler, but only the last TickCount setting will be in effect. TickCount affects only the specified registered timer handler. It has no effect on other timer handlers. Timer handlers are responsible for saving and restoring registers on entry to and exit from this service. A maximum of 32 timer handlers are available in the system.

DS should be set to the physical device driver's data segment. If the physical device driver did a call to PhysToVirt referencing the DS register, it will restore DS to the original value.

Unlock

This service unlocks a locked memory segment.

Calling Sequence

```
MOV  BX,lock_handle_low    ; Handle for segment returned by Lock
MOV  AX,lock_handle_high   ;
MOV  DL,DevHlp_Unlock
```

```
CALL [Device_Help]
```

Results

'C' Clear if segment unlocked.

'C' Set if error.
AX = Error code.

Remarks: None.

UnPhysToVirt

This service is required to mark completion of address conversion from PhysToVirt.

Remarks: This function still exists for compatibility reasons but is no longer operational.

UnSetIRQ

This service removes the current hardware interrupt handler.

Calling Sequence

```
MOV  BX,IRQnum      ; IRQ Interrupt number (0 - F)
MOV  DL,DevHlp_UnSetIRQ
CALL [Device_Help]
```

Results

'C' Set if the caller is not the owner or one of the owners of the IRQ.

Remarks: DS must point to the data segment of the physical device driver on entry.

VerifyAccess

This service is used to verify that the user process has the correct access rights for the memory that it passed to the physical device driver. If the process does not have the needed access rights to the memory, then it is terminated. If it does have the needed access rights, these rights are guaranteed to remain valid until the physical device driver exits its strategy routine.

Calling Sequence

```
MOV  AX,Segment_@      ; Selector or segment
MOV  CX,MemLength      ; Length of memory area in bytes (0=64KB)
MOV  DI,Mem_Offset     ; Offset to memory area
MOV  DH,TypeOfAccess   ; Verify access for...
                        ; 0 = read access
                        ; 1 = read/write access

MOV  DL,DevHlp_VerifyAccess

CALL [Device_Help]
```

Results

'C' Clear if successful.
Access is verified.

'C' Set if error (if the access attempt failed).

Remarks: A physical device driver can receive addresses to memory as part of a generic IOCTL request from a process. Because the operating system cannot verify addresses imbedded in the IOCTL command, the physical device driver must request verification in order to prevent itself from accidentally erasing memory on behalf of a user process. If the verification test fails, then VerifyAccess terminates the process.

After the physical device driver verifies that the process has the needed access rights to the addresses of interest, it does not need to repeat the verification until it yields the CPU. When the device driver yields the CPU, all address access verifications it has done become unreliable, except for segments that have been locked. The physical device driver could yield the CPU by accessing a *not-present-segment* exiting its strategy routine, or by calling a DevHlp service that yields while performing the service.

If the physical device driver plans to lock the segment into memory (by using the DevHlp, Lock), it should perform the lock before the call to VerifyAccess. If the selector is invalid, the Lock DevHlp returns an error. The physical device driver can then call VerifyAccess. VerifyAccess terminates the process, if the address lacks sufficient access permission for the requested I/O operation.

Note: If the physical device driver successfully calls the Lock DevHlp before calling VerifyAccess, and the VerifyAccess call terminates the application, the physical device driver must unlock the segment (using the DevHlp, Unlock) before returning.

VideoPause

This service is called by physical device drivers when the controller reports a DMA overrun. VideoPause starts or stops high-priority threads. This halts threads using the CPU for video transfers, which allows the diskette DMA to complete termination properly.

Calling Sequence

```
MOV  AL,VideoPause_Flag      ; VideoPause flag. 0 = Off, !0 = On
MOV  DL,DevHlp_VideoPause

CALL [Device_Help]
```

Results

'C' Clear if successful.

'C' Set if error.

Remarks: VideoPause can be called at initialization time, task time, or interrupt time. This service is used after a retry has failed. To avoid impairing system performance, VideoPause is turned on only long enough to accomplish the transfer. If multiple physical device drivers have turned VideoPause *on*, it is not turned *off* until all of them have turned it *off*.

VirtToLin

This service converts a selector:offset pair into a linear address.

Calling Sequence

```
MOV  AX,Selector      ; Selector in Selector:Offset
MOV  ESI,Offset       ; Offset in Selector:Offset
MOV  DL,DevHlp_VirtToLin

CALL [Device_Help]
```

Results

'C' Clear if Linear address obtained.
EAX = Linear address.

'C' Set if linear address not obtained.
EAX = Error code.

Remarks: None.

VirtToPhys

This service converts a selector:offset pair to a 32-bit physical address.

Calling Sequence

```
LDS  SI,address      ; Virtual address:
                        ; selector:offset
MOV  DL,DevHlp_VirtToPhys
CALL [Device_Help]
```

Results

'C' Clear if successful.
AX:BX = Physical address: 32-bit number.

Remarks: If the segment is not already known to be locked, the virtual address should be locked using the DevHlp, Lock, prior to calling this service. VirtToPhys is typically used to convert a virtual address, supplied to the physical device driver by a process (through a generic IOCTL), to a physical memory address so that the memory can be accessed at interrupt time.

VMAAlloc

This service allocates virtual memory and, depending on the value of a flag, either commits physical storage or maps virtual memory to a given physical address.

Calling Sequence

```

MOV  ECX,Size           ; Size of object in bytes
MOV  EDI,OFFSET PhysAddr ; Physical address to be mapped (Not used if set to -1)
                                ; (Used only if Flags is set to 08H or 16H)
MOV  EAX,Flags          ; Flags used for allocation request
                                ; If bit 0 is set (000000001B), the object will be allocated
                                ; below the 16 MB address line.
                                ; If bit 1 (000000010B) is set, the object needs to be
                                ; fixed in memory at all times. If this bit is clear,
                                ; then the object may be moved or paged out as necessary.
                                ; If bit 2 (000000100B) is set, then swappable memory will be
                                ; allocated for the object. If this bit is clear, then the
                                ; memory will either movable or fixed depending on the setting
                                ; of bit 1. If this bit is set, bit 1 must be clear.
                                ; If bit 3 (000001000B) is set, the object needs to be allocated
                                ; in contiguous memory. If this bit is clear then the pages
                                ; can be discontinuous in physical memory. In order to
                                ; request contiguous memory the physical device driver must have
                                ; also requested fixed memory (bit 1 must also be set).
                                ; If bit 4 (000010000B) is set, then a linear address mapping
                                ; will be obtained for the physical address passed in the
                                ; PhysAddr pointer.
                                ; If bit 5 (000100000B) is set, the linear address returned
                                ; will be in process address range. If this bit is clear,
                                ; the allocation/mapping will be done in global address range,
                                ; that is, accessible out of the current process' context.
                                ; If bit 6 (001000000B) is set, the allocated memory can be
                                ; registered under screen group switch control. This flag
                                ; is valid only if mapping is in process address range as
                                ; well (bit 5 must be set).
                                ; Bit 7 (010000000B) is reserved and should be set to 0.
                                ; If bit 8 (100000000B) is set, the memory will only be reserved.
                                ; No commitment will be done and any attempt to access reserved
                                ; but not committed memory will cause a fatal page fault.
                                ; All other bits must be Clear.
```

```
MOV  DL,DevHlp_VMAAlloc
```

```
CALL [Device_Help]
```

Results

'C' Clear if object allocated.
EAX = Linear address of object.

'C' Set if error.
EAX = Error code.
Possible errors:
ERROR_INVALID_PARAMETER

Remarks: This function obtains a global, Ring 0, linear mapping to a block of memory. The physical address of the memory can be specified for non-system memory, or the system will allocate the block from general system memory. A linear address is returned to address the memory. For contiguous fixed allocation requests, the physical address is also returned.

Virtual memory is allocated in global (system) address space, unless private process space is requested. Memory requested in process space can only be swapped. If requested, memory allocated in process space can be registered under screen group switch control. In that case, a task is denied write access to this memory unless it is in foreground.

Bit 0 is used by physical device drivers, which cannot support more than 16MB addresses. If the physical device driver requests memory below 16MB, the memory must also be resident at all times. If bit 4 is set, virtual memory is mapped to a given physical address. In this case, the physical memory must be fixed or locked (that is, bit 1 should be set as well). This could be used for non-system memory like Video buffers. If it is used for system memory, it is the responsibility of the device driver to ensure that the physical pages corresponding to the PhysAddr never move or become invalid. Memory or mapping allocated with bit 6 set is invalid when the process is not in foreground. If bit 8 is set, the linear address returned is page aligned. The size requested is rounded up to the nearest page boundary. All other allocations can return byte-granular size and addresses.

To use the memory management DevHlp services to give a process access to a video buffer, perform the following steps:

1. Call VMAAlloc to obtain a linear address in the process's address space for the physical buffer. Register it under screen group switch control to invalidate access to it if the task is not current.
2. If the process needs to access the memory with a tiled Local Descriptor Table (LDT) selector, allocate with a selector map request, and convert the linear address returned by VMAAlloc to an LDT selector:offset with the FlatToSel macro.
3. Call VMFree to remove addressability to the buffer.

VMFree

This service frees memory allocated with VMAlloc, or a mapping created by VMProcessToGlobal or VMGlobalToProcess.

Calling Sequence

```
MOV  EAX,Linear_Address  ; Linear address of the region to be freed
MOV  DL,DevHlp_VMFree
CALL [Device_Help]
```

Results

'C' Clear if memory freed.

'C' Set if error.

EAX = Error code.

Possible errors:

ERROR_ACCESS_DENIED

ERROR_INVALID_PARAMETER

Remarks: All memory or mapping allocated by the physical device driver must be released before device driver termination.

VMGlobalToProcess

This service maps an address in the system region of the global address space into an address in the address space of the current process.

Calling Sequence

```
MOV  EBX,LinearAddress    ; Linear Address in global address space
MOV  ECX,Length           ; Length (in bytes) of the memory object to be mapped
MOV  EAX,ActionFlags      ; Actions to perform on user region
                                ; If bit 0 (0001B) is set, the process context is mapped for
                                ;   read/write access to the object. If this bit is clear,
                                ;   then read only permission is given
                                ;   (Note: Some hardware may not support read only mapping).
                                ; If bit 1 (0010B) is set, 16-bit selectors are allocated
                                ;   to map the 32-bit region at 64KB boundaries similar
                                ;   to DosAllocHuge.
                                ; If bit 2 (0100B) is set, the mapping is tracked for validation
                                ;   and invalidation of screen buffers.
                                ; If bit 3 (1000B) is set, the memory is allocated on a 4MB
                                ;   boundary which improves performance on the screen group
                                ;   validation/invalidation. This also reserves the entire 4MB
                                ;   region starting at the 4MB boundary.
                                ; All other bits must be clear.

MOV  DL,DevHlp_VMGlobalToProcess

CALL [Device_Help]
```

Results

'C' Clear if conversion performed.
EAX = Linear address within process's address space.

'C' Set if conversion not performed.
EAX = Error code.

Remarks: The mapping created by this function must be released with VMFree. The address range must not cross object boundaries. The address space used in this service is of the current process. In Figure 17-2 on page 17-102, VMGlobalToProcess maps the memory allocated with VMAlloc into Process A's private address space.

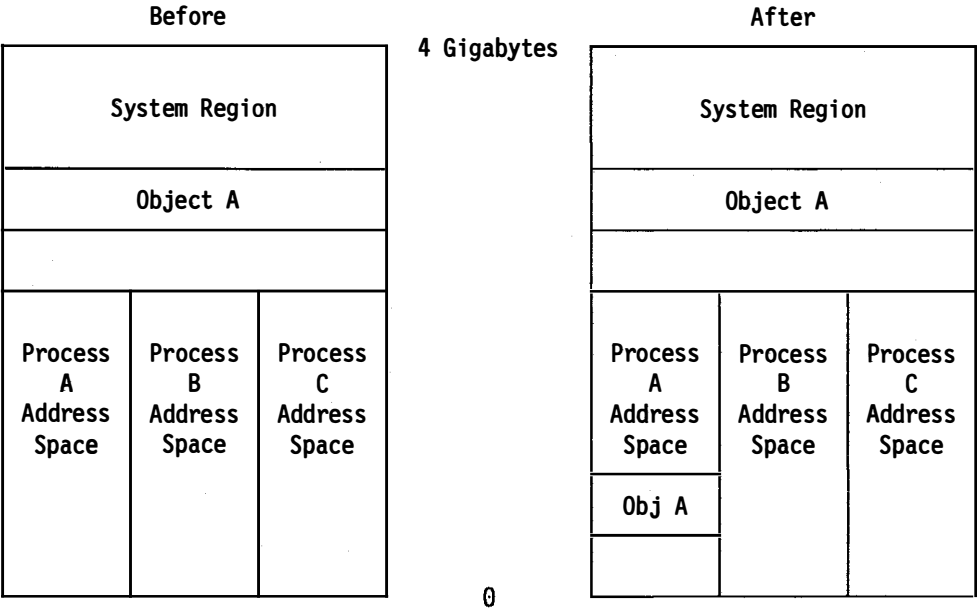


Figure 17-2. Mapping Performed by VMGlobalToProcess.

VMLock

This service verifies accessibility to a region of memory, and locks the memory region into physical memory. If the region is unavailable, the caller must specify whether VMLock should block until the region is available and locked, or return immediately.

Calling Sequence

```

MOV  EBX,LinearAddress    ; Linear Address of region to lock
MOV  ECX,Length           ; Count of bytes to be locked
MOV  EDI,OFFSET pPageList ; Flat pointer to array of PageList_s structures filled by the call
MOV  ESI,OFFSET pLockHandle ; Address of the 12-byte variable the lock handle will be placed in
MOV  EAX,ActionFlags      ; Actions to be performed for lock
                                ; If bit 0 (0000001B) is set, the lock returns if the requested pages
                                ;   are not immediately available. If the bit is clear, the call
                                ;   blocks until the pages are available.
                                ; If bit 1 (0000010B) is set, the pages that are to be locked must be
                                ;   physically contiguous. This should only be used by device drivers
                                ;   that are about to perform contiguous DMA operations on the object.
                                ; If bit 2 (0000100B) is set, the locked pages must be below the 16MB
                                ;   address line. If this bit is clear then the pages can be locked
                                ;   anywhere in memory.
                                ; If bit 3 (0001000B) is set, then the physical device driver
                                ;   plans to write in the region of the segment.
                                ;   This will cause the dirty bit to be set in the page
                                ;   table entries since the physical device driver may be doing
                                ;   DMA to the pages which does not set this bit.
                                ; If the dirty bit is not set when writing to a page
                                ;   then the data written to the page may be lost.
                                ; If bit 4 (0010000B) is set, then the lock is needed for a long-term duration.
                                ;   This parameter would be used if the physical device driver intends to keep
                                ;   the lock on the object for greater than two seconds.
                                ; If bit 5 (0100000B) is set, the region will be
                                ;   'verify only' locked. This means that the page
                                ;   table mapping of this region cannot be removed or
                                ;   destructively modified (No free or decommit allowed
                                ;   in the region, cannot remove write permission).
                                ;   However the region will not be physically locked
                                ;   in memory, the physical pages will retain their
                                ;   original status (swappable, discardable, etc...).
                                ;   This lock also does not force any pages to be present.
                                ;   Passing the returned lock handle info back to VMUnLock
                                ;   will release 'verify only' lock. The flags in bit
                                ;   positions 1, and 2 are invalid with this request.
                                ;   No physical addresses will be returned in 'verify' lock
                                ;   requests. Verify locks can be specified for short or
                                ;   long term. If bit 3 is set along with this bit, the
                                ;   memory will be verified for read/write access. If bit 3
                                ;   is clear, only the read permission will be verified.
                                ; All other bits must be clear.

MOV  DL,DevHlp_VMLock

CALL [Device_Help]

```

Each PageList_s structure describes a single physically contiguous subregion of the physical that was locked. The format of the PageList_s structure is:

```

PageList_s struc
    pl_PhysAddr DD ? ; Physical address of first byte in this sub-region
    pl_cb       DD ? ; Number of contiguous bytes starting at pl_PhysAddr
PageList_s ends

```

Results

'C' Clear if pages locked.
EAX = If caller requested, the number of elements in the PageList array.

'C' Set if segment unavailable.
EAX = Error code.

Remarks: Use of short-term locks for greater than two seconds can prevent an adequate amount of pages being available for system use. Under these circumstances, *a system halt could occur*. If satisfying the lock request will reduce the amount of free memory in the system below a predetermined minimum, both short-term and long-term locks can fail .

Address verification is done automatically with every VMLock request. Locking down memory in fixed physical addresses is done only if the *verify only* bit is not set. It is the responsibility of the physical device driver to ensure that enough entries have been reserved for the range of memory being locked (possibly one entry per page in the range plus one more, if the region does not begin on a page boundary). If this pointer contains the value *-1*, then no physical addresses are returned. This parameter must be *-1* for verify locks.

Since locking occurs on a per page basis, the VMLock service routine will round LinearAddress down to the nearest page boundary. If physically contiguous locking is requested, Length cannot exceed 64KB, otherwise an error is returned. Because locking occurs on a per page basis, the combination of LinearAddress + Length is rounded up to the nearest page boundary.

VMProcessToGlobal

This service converts an address in the current process address space to an address in the system region of the global address space.

Calling Sequence

```
MOV  EBX,LinearAddress    ; Linear Address within process address space
                          ; that is to be mapped into a global context;
                          ; must be on a page boundary
MOV  ECX,Length           ; Length of request in bytes
MOV  EAX,ActionFlags      ; Action to perform
                          ; If bit 0 (00001B) is set, the mapping created will be
                          ; writable. If this bit is clear, the mapping will
                          ; be read only
                          ; (Note: Some hardware may not support read only mapping).
                          ; All other bits must be clear.
```

```
MOV  DL,DevHlp_VMProcessToGlobal
```

```
CALL [Device_Help]
```

Results

'C' Clear if conversion performed.
EAX = Global offset to region of memory.

'C' Set if conversion not performed.
EAX = Error code.

Remarks: The address range must be on a page boundary, and must not cross object boundaries. This call copies the linear mapping from the process's address space to the system-shared address space, this allows the physical device driver to access the data independent of the context of the current process.

Figure 17-3 shows the mapping that is performed when a physical device driver calls VMProcessToGlobal.

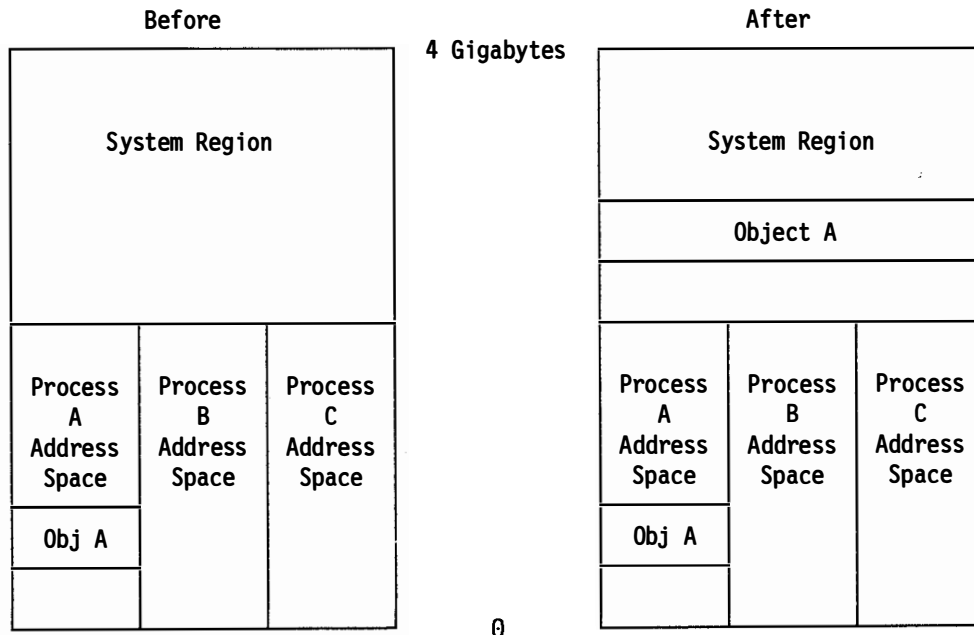


Figure 17-3. Mapping Performed by VMProcessToGlobal.

The following steps show to use the DevHlp services to gain interrupt time access to the buffer of a process:

1. If the user's buffer was allocated in the Local Descriptor Table (LDT) tiled region, use the `SelfToFlat` macro to convert the address to a linear address.
2. Call `VMLock` to verify the address and to lock the range of memory needed into physical memory.
3. Call `VMProcessToGlobal` to map the private address of a process into global address space. See the previous figure for more details on the mapping performed.
4. If the physical device driver requests it, an array of physical addresses corresponding to the locked region is returned.
5. Access the memory using the FLAT selector and its linear address as returned by `VMProcessToGlobal`.
6. Call `VMFree` to remove global mapping to process address space.
7. Call `VMUnlock` to unlock the object.

VMSetMem

This service commits and decommits physical storage, or changes the type of committed memory prereserved with the VMAlloc DevHlp service. The address range specified must not cross object boundaries. The range must be entirely of uniform type, that is, all decommitted (invalid), all swappable, or all resident. The range to be decommitted must be entirely precommitted.

Calling Sequence

```
MOV    EBX,LinearAddress    ; Linear address of the region; must be page aligned;
MOV    ECX,Size             ; Size of the region (in bytes) must be page granular
MOV    EAX,Flags            ; Flags used in the request
                                ; If bit 0 is set (00000001B), the address range is to be
                                ; decommitted. If this bit is set, the range must be
                                ; entirely committed.
                                ; If bit 1 is set (00000010B), the address range is to be made
                                ; resident. The entire address range must be decommitted, all
                                ; resident or all swappable. If the address range is already
                                ; resident, the function will return with 'C' (the carry flag)
                                ; cleared, indicating success, without taking any action.
                                ; If the address range is swappable, it will be changed to
                                ; resident. If it is uncommitted, it will be committed as
                                ; resident memory.
                                ; If bit 2 is set (00000100B), the address range is to be made
                                ; swappable. The entire address range must be all decommitted,
                                ; all resident, or all swappable. If the range is already
                                ; swappable, the function will return with 'C' (the carry flag)
                                ; cleared, indicating success, without taking any action.
                                ; If the address range is resident, it will be made swappable.
                                ; If it is uncommitted, it will be committed as swappable memory.
                                ; All other bits are clear.
```

```
MOV    DL,DevHlp_VMSetMem
CALL   [Device_Help]
```

Results

```
'C' Clear if successful.
    EAX = 0.

'C' Set if error.
    EAX = Error code.
        Possible errors:
            ERROR_ACCESS_DENIED
            ERROR_CROSSES_OBJECT_BOUNDARY
            ERROR_INVALID_PARAMETER
```

Remarks: The entire region, LinearAddress + Size, must lie within a memory object previously allocated with VMAlloc, which has the Reserve Only option set.

VMUnlock

This service unlocks a previously locked memory range.

Calling Sequence

```
MOV  ESI,OFFSET LockHandle ; Flat pointer to the Lock handle returned from VMLock
MOV  DL,DevHlp_VMUnlock

CALL [Device_Help]
```

Results

'C' Clear if pages unlocked.

'C' Set if error.

EAX = Error code.

Possible errors:

- ERROR_INVALID_HANDLE
- ERROR_INVALID_PARAMETER
- ERROR_NOT_LOCKED

Remarks: A successful call to this function can modify the caller's lock handle.

Yield

This service yields the CPU to a scheduled thread of equal, or higher, priority.

Calling Sequence

```
MOV    DL,DevHlp_Yield
```

```
CALL   [Device_Help]
```

Results: None.

Remarks: The OS/2 operating system is designed so that the CPU is never scheduled preemptively, while in kernel mode. In general, the kernel either performs its job and exits quickly, or it blocks waiting for I/O (or occasionally a resource). It is not necessary for the device driver to issue both a Yield and a TCYield; the Yield function is a superset of the TCYield function.

Physical device drivers are the one part of the kernel that can use a large amount of CPU time, particularly those that perform program I/O on long strings of data, or that poll the device. These physical device drivers should check the Yield flag periodically, and call the Yield function to yield the CPU if another process needs it. Most of the time the context won't switch. Yield switches context only if an equal, or higher, priority thread is scheduled to run. The address of the Yield flag is obtained from the GetDOSVar service, which is used to get the addresses of kernel variables. See "GetDOSVar" on page 17-36.

For performance reasons, the physical device driver should check the Yield flag once every 3 milliseconds. If the flag is set, then the physical device driver should call Yield. Because the physical device driver can relinquish control of the CPU to another thread, it must not assume that the state of the interrupt flag will be preserved across a call to Yield.

Chapter 18. Generic IOCTL Commands

OS/2 device drivers are used to access the I/O hardware. The *IOctl* functions provide a method for an application, or subsystem, to send device-specific control commands to a physical device driver. These IOCTLs are subfunctions that are issued through the DosDevIOctl API function request. The DosDevIOctl function request can only be used by OS/2 applications; the INT 21H IOCTL request can only be used by DOS applications.

The category and function fields are as follows. Each code is contained in a byte:

Category	Code
0xxx xxxx	OS/2 defined
1xxx xxxx	User defined
__xxx xxxx	Code.
Function	Code
0xxx xxxx	Return error, if unsupported
1xxx xxxx	Ignore, if unsupported
x0xx xxxx	Intercepted by the OS/2 operating system
x1xx xxxx	Passed to driver
xx0x xxxx	Sends data and commands to device
xx1x xxxx	Queries data and information from device
___x xxxx	Subfunction.

Notice that the *send/query* data bit is intended only to regularize the function set; it plays no critical role. Some functions can contain both command and query elements. Such commands are defined as *sends data*.

Generic IOCTL Example: The DosDevIOctl function sends the request to the physical device driver request packet. The physical device driver receives the request packet, and looks for the Command Code (Command 16 is the generic IOCTL command) to identify the request. Notice that each device driver can define the structure of the Data Packet and the Parameter Packet, but all device drivers use the same request header.

The calling sequence for DosDevIOctl is shown below:

```
EXTRN  DosDevIOctl:Far

PUSH@  OTHER  Data      ; Data Packet address
PUSH@  OTHER  ParmList  ; Parameter Packet address
PUSH    WORD   Function  ; Function code
PUSH    WORD   Category  ; Category code
PUSH    WORD   DevHandle ; User's device driver file handle

CALL    DosDevIOctl
```

DosDevIOctl2 performs the same function as DosDevIOctl, and also provides Length fields for the Data and Parameter List buffers. These Length fields should be passed to the Device Helper service, "VerifyAccess" on page 17-94, when the physical device driver is determining whether it has access to the application's Data and Parameter List buffers. The Length fields also tell physical Network device drivers the amount of data residing in these buffers for the transfer to other network nodes.

generic IOCTL commands

The calling sequence for DosDevIOCtl2 is shown below:

```
EXTRN  DosDevIOCtl2:Far

PUSH@  OTHER  Data      ; Data Packet address
PUSH   WORD   DataLength ; Data Packet length
PUSH@  OTHER  ParmList   ; Parameter Packet address
PUSH   WORD   ParmLength ; Parameter Packet length
PUSH   WORD   Function   ; Function code
PUSH   WORD   Category   ; Category code
PUSH   WORD   DevHandle  ; User's device driver file handle

CALL   DosDevIOCtl2
```

Generic IOCTL Function Table

The list of categories and functions for the generic IOCTL requests are as follows:

Category	Function	Description
01		Serial Device Control
	14H	Reserved
	34H	Reserved
	41H	Set Bit Rate
	42H	Set Line Characteristics (stop, parity, data bits)
	43H	Extended Set Bit Rate
	44H	Transmit Byte Immediate
	45H	Set Break OFF
	46H	Set Modem Control Signals
	47H	Behave As If XOFF Received (stop transmit)
	48H	Behave As If XON Received (start transmit)
	49H	Reserved
	4BH	Set Break ON
	53H	Set Device Control Block (DCB) Parameters
	54H	Set Enhanced Mode Parameters
	61H	Query Current Bit Rate
	62H	Query Line Characteristics
	63H	Extended Query Bit Rate
	64H	Query COM Status
	65H	Query Transmit Data Status
	66H	Query Modem Control Output Signals
	67H	Query Current Modem Input Signals
	68H	Query Number of Characters in Receive Queue
	69H	Query Number of Characters in Transmit Queue
	6DH	Query COM Error
	72H	Query COM Event Information
	73H	Query Device Control Block (DCB) Parameters
	74H	Query Enhanced Mode Parameters
02		Reserved
03		Pointer Draw Control
	72H	Query Pointer Draw Address
03		Video Control
	73H	Initialize Call Vector Table
03		Screen Control
	70H	Allocate an LDT Selector
	71H	Deallocate an LDT Selector

generic IOCTL function table

Category	Function	Description
	74H	ABIOS Pass-Through
	75H	Allocate an LDT Selector with Offset
	76H	Allocate an LDT Selector with Background Validation Options
04		Keyboard Control
	50H	Set Code Page
	51H	Set Input Mode (Default ASCII)
	52H	Set Interim Character Flags
	53H	Set Shift State
	54H	Set Typematic Rate and Delay
	55H	Reserved
	56H	Set Session Manager Hot Key
	57H	Set KCB
	58H	Set Code Page Number
	59H	Set Read/Ppeek Notification
	5AH	Alter Keyboard LEDs
	5BH	Reserved
	5CH	Set NLS and Custom Code Page
	5DH	Create New Logical Keyboard
	5EH	Destroy Logical Keyboard
	71H	Query Input Mode
	72H	Query Interim Character Flags
	73H	Query Shift State
	74H	Read Character Data Records
	75H	Peek Character Data Record
	76H	Query Session Manager Hot Key
	77H	Query Keyboard Type
	78H	Query Code Page Number
	79H	Translate Scan Code to ASCII
	7AH	Query Keyboard Hardware ID
	7BH	Query Keyboard Code Page Support Information
05		Parallel Port Control
	42H	Set Frame Control (CPL, LPI)
	44H	Set Infinite Retry
	45H	Reserved
	46H	Initialize Parallel Port
	47H	Reserved
	48H	Activate Font
	4BH	Reserved
	4CH	Reserved

Category	Function	Description
	4DH	Set Print-Job Title
	4EH	Set Parallel Port IRQ Time-Out Value
	4FH	Reserved
	62H	Query Frame Control
	64H	Query Infinite Retry
	66H	Query Parallel Port Status
	67H	Reserved
	69H	Query Active Font
	6AH	Verify Font
	6BH	Reserved
	6CH	Reserved
	6DH	Reserved
	6EH	Query Parallel Port IRQ Time-Out Value
	6FH	Reserved
06		Light Pen Control
07		Mouse Control
	50H	Reserved
	51H	Notification of Display Mode Change
	52H	Reserved
	53H	Reassign Current Mouse Scaling Factors
	54H	Assign New Mouse Event Mask
	55H	Reserved
	56H	Set Pointer Shape
	57H	Unmark Collision Area
	58H	Mark Collision Area
	59H	Specify/Replace Pointer Screen Position
	5AH	Set OS/2-Mode Pointer Draw Device Driver Address
	5BH	Reserved
	5CH	Set Current Physical Mouse Device Driver Status Flags
	5DH	Notification of Mode Switch Completion
	60H	Query Number of Mouse Buttons Supported
	61H	Query Mouse Device Motion Sensitivity
	62H	Query Current Physical Mouse Device Driver Status Flags
	63H	Read Mouse Event Queue
	64H	Query Current Event Queue Status
	65H	Query Current Mouse Event Mask
	66H	Query Current Mouse Scaling Factors
	67H	Query Current Pointer Screen Position
	68H	Query Current Pointer Shape

generic IOCTL function table

Category	Function	Description
	69H	Reserved
	6AH	Query Physical Mouse Device Driver Level/Version
	6BH	Query Pointing Device ID
08		Logical Disk Control
	00H	Lock Drive
	01H	Unlock Drive
	02H	Redetermine Media (end format)
	03H	Set Logical Map
	04H	Begin Format
	20H	Block Removable
	21H	Query Logical Map
	22H	Reserved
	43H	Set Device Parameters
	44H	Write Track
	45H	Format and Verify Track
	5EH	Reserved
	5FH	Reserved
	60H	Query Media Sense
	63H	Query Device Parameters
	64H	Read Track
	65H	Verify Track
09		Physical Disk Control
	00H	Lock Physical Drive
	01H	Unlock Physical Drive
	44H	Physical Write Track
	63H	Query Physical Device Parameters
	64H	Physical Read Track
	65H	Physical Verify Track
10		Character Device Monitor Control
	40H	Register Monitor
11		General Device Control
	01H	Flush Input Buffer
	02H	Flush Output Buffer
	41H	System Notifications for Physical Device Drivers
	60H	Query Monitor Support
12-127		Reserved Category Codes

Category 1 ASYNC (RS232-C) Control IOCTL Commands

Whenever an IOCTL command calls for a NULL pointer, it is the responsibility of the application to set one up for the appropriate Parameter or Data Packet pointer before calling the physical device driver. IOCTLs can be interpreted differently by future releases, if the pointer is not a NULL pointer. If a NULL pointer is called for and it is not received by the device driver, it is considered an invalid Parameter or Data Packet value.

The physical device driver services each communications port (COM1, COM2, and so forth) independently. IOCTLs issued to the physical device driver for a given port have no effect on any other communications ports that the physical device driver is servicing. The application cannot assume a given timing relationship between when the IOCTLs are executed, and when data is received or transmitted by the ASYNC hardware. Data Carrier Detect (DCD) is the same signal as Receiver Line Signal Detect (RLSD).

The following is a summary of the Category 1 IOCTL Commands:

Function	Description
14H	Reserved
34H	Reserved
41H	Set Bit Rate
42H	Set Line Characteristics (stop, parity, data bits)
43H	Extended Set Bit Rate
44H	Transmit Byte Immediate
45H	Set Break OFF
46H	Set Modem Control Signals
47H	Behave as if XOFF Received (stop transmit)
48H	Behave as if XON Received (start transmit)
49H	Reserved
4BH	Set Break ON
53H	Set Device Control Block (DCB) Parameters
54H	Set Enhanced Mode Parameters
61H	Query Current Bit Rate
62H	Query Line Characteristics
63H	Extended Query Bit Rate
64H	Query COM Status
65H	Query Transmit Data Status
66H	Query Modem Control Output Signals
67H	Query Current Modem Input Signals
68H	Query Number of Characters in Receive Queue
69H	Query Number of Characters in Transmit Queue
6DH	Query COM Error
72H	Query COM Event Information
73H	Query Device Control Block (DCB) Parameters
74H	Query Enhanced Mode Parameters

Function 41H – Set Bit Rate

This function sets the bit rate.

Parameter Packet Format

Field	Length
Bit Rate	WORD

Bit Rate The Bit Rate field is a binary integer representing the actual bit rate (bits-per-second) that the physical device driver uses to set the bit rate of the COM device. For bit rates up to 19200 bps, the physical device driver sets the rate only if the hardware can support the rate within .01% margin of error. The exceptions are for 110 and 2000 bps, which can have an error of up to .026% and .69%, respectively. In all other cases, if the requested rate cannot be achieved by the hardware within .01% tolerance, the IOctI fails, with an ERROR_INVALID_PARAMETER return code.

For bit rates beyond 19200 bps, the physical device driver does not check the .01% margin of error. It is the user's responsibility to maintain the bit rate tolerance. After calling Function 41H to set a bit rate higher than 19200 bps, the user calls "Function 61H – Query Bit Rate" to check the actual bit rate set on a COM port.

The recommended bit rate values are:

Bit Rate Values	Bit Rate Values
-----	-----
110	3600
150	4800
300	7200
600	9600
1200	19200
1800	38400
2000	57600
2400	

Data Packet Format: None. Packet pointer must be NULL.

Returns: If the call is made with invalid Parameter Packet values or an invalid Data Packet pointer, a *general failure* error is reported.

Remarks: If a general failure error is not returned, the physical device driver performs the action described in the Bit Rate field. An OPEN request packet does not cause the physical device driver to change the bit rate from its previous value. The initial value is 1200 bps.

The COM device hardware determines which bit rates can be supported on a given system. The range of bit rates supported by Function 41H is 2 bps to 19200 bps for COM ports on conventional serial devices. For COM ports on enhanced serial devices, the range is 10 bps to 57600 bps. A call with a value beyond this range fails, with the ERROR_INVALID_PARAMETER return code.

Function 42H – Set Line Characteristics

This function sets the line characteristics (stop bits, parity, data bits).

Parameter Packet Format

Field	Length
Data Bits	BYTE
Parity	BYTE
Stop Bits	BYTE

Data Bits Has the following value and meaning:

00H-04H	Reserved
05H	5 data bits
06H	6 data bits
07H	7 data bits (initial value)
08H	8 data bits
09H-FFH	Reserved.

Parity Has the following value and meaning:

00H	No parity
01H	Odd parity
02H	Even parity (initial value)
03H	Mark parity (parity bit always 1)
04H	Space parity (parity bit always 0)
05H-FFH	Reserved.

Stop Bits Has the following value and meaning:

00H	1 stop bit (initial value)
01H	1.5 stop bits (valid with 5-bit WORD length only)
02H	2 stop bits (not valid with 5-bit WORD length)
03H-FFH	Reserved.

Data Packet Format: None. Packet pointer must be NULL.

Returns: If the call is made with invalid Parameter Packet values or an invalid Data Packet pointer, a *general failure* error is reported, and the line characteristics are not changed for any parameters that were valid.

Remarks: If a general failure error is not returned, the physical device driver sets the line characteristics as defined. An OPEN request packet does not cause the physical device driver to change the line characteristics from its previous values.

If the WORD length is less than 8 bits, the appropriate high-order bits for received data are 0, when placed in the receive queue and operated on by the physical device driver (for example, XON/XOFF checking, null stripping). This only applies to data that is received after the command is operated on by the physical device driver. Data already in the physical device driver receive queue is not affected in any way by a change in the WORD length.

If the WORD length is less than 8 bits, the physical device driver does not automatically truncate control/transmit data that is passed by the application. No error is reported by the device driver, if transmit or control data given to it has high-order bits of non-zero value.

For example, if the physical device driver is told that the WORD length is 7 bits, and the XOFF character is 80H, then the physical device driver is not able to recognize the XOFF character, if Automatic Transmit Flow Control is enabled. If the error substitution character is set to 80H by the application, with a WORD length of 7 currently active, the device driver still places an 80H in the receive queue. It is the responsibility of the application to maintain consistency between the requested WORD length for the COM device, and the requests that the application makes of the physical device driver.

Function 43H – Extended Set Bit Rate

This function sets the bit rate in doublewords to cover bit rates higher than 19200 bps.

Parameter Packet Format

Field	Length
Bit Rate	DWORD
Fraction	BYTE

Bit Rate The Bit Rate field is a binary integer representing the actual bit rate (bits-per-second) that the physical device driver uses to set the bit rate of the COM device. For a COM port that is supported by Enhanced UART, or compatible serial communication devices, the range is 10 bps – 345600 bps. The user can get the minimum and maximum bit rates, supported by the physical device driver on a COM port, by using “Function 63H – Extended Query Bit Rate” on page 18-41.

Note: Due to system overhead, and the physical limits on the hardware cables, actual bit rates might be restricted.

When this function is used for a COM port (which has no support of the Enhanced UART), or compatibles, then the bit range supported by the device driver is from 2 bps – 19200 bps. A call with a bit rate out of range fails, with the `ERROR_INVALID_PARAMETER` return code.

Fraction This field is a binary integer value that represents the fraction of the bit rate to set. It is used for setting a very precise bit rate. In most cases, this field is set to 0.

If the Fraction field needs to be set, the user must understand how bit rates are handled in the system. Bit rate is calculated using the following formula:

$$\text{DIVISOR COUNT} = \text{BAUD CLOCK} / \text{SCALER} / \text{BINARY BIT RATE}$$

For the Enhanced UART, the BAUD CLOCK is 22.1184 MHz., and the SCALER is 32. The DIVISOR COUNT is a 16-bit value that is loaded into the ASYNC device’s divisor latch to generate the final bit rate clock. Any fractional amount is rounded to the nearest divisor count to get the closest bit rate. This integer is put into the following formula to get the actual output bit rate value:

$$\text{OUTPUT BIT RATE} = \text{BAUD CLOCK} / \text{SCALER} / \text{DIVISOR COUNT}$$

For the value of the Fraction field, the resultant fraction obtained from the above formula is multiplied by 256. When setting a bit rate, the physical device driver passes the bit rate and fraction directly to ABIOS.

For bit rates up to 19200 bps, the physical device driver sets the rate, only if the hardware can support the rate within .01% margin of error to maintain compatibility. The exceptions are for 110 and 2000 bps, which can have an error of up to .026% and .69%, respectively. In all other cases, if the requested rate cannot be achieved by the hardware within .01% tolerance, the IOctI fails, with an `ERROR_INVALID_PARAMETER` return code.

For bit rates beyond 19200 bps, the physical device driver does not check the .01% margin of error. It is the user’s responsibility to maintain the bit rate tolerance. After calling Function 43H, the user should call Function 63H to check the actual bit rate set on a COM port.

The recommended bit rate values are:

Bit Rate Values	Bit Rate Values				
110	7200				
150	9600				
300	19200				
600	38400				
1200	57600				
1800	76800	(use IOctl Function 74H)			
2000	115200	"	"	"	"
2400	230400	"	"	"	"
3600	345600	"	"	"	"
4800					

Data Packet Format: None. Packet pointer must be NULL.

Returns: If the call is made with invalid Parameter Packet values or an invalid Data Packet pointer, a *general failure* error is reported.

Remarks: If a general failure error is not returned, the physical device driver performs the action described in the Bit Rate field. An OPEN request packet does not cause the physical device driver to change the bit rate from its previous value. The initial value is 1200 bps.

The physical ASYNC device driver is designed to handle high bit rates under optimum conditions. To achieve successful data transfer operations at high bit rates, and to obtain high data throughput, the following conditions must be met:

- The connecting cables must be free from electrical noise, and maintain the voltage conforming to RS232-C standard. IBM Personal Computer* Communication adapter cables (P/N 6323741 or 1502067) can guarantee this for up to 20 feet.
- DMA capability must be available.
- Little system overhead. Operations at high bit rates require many CPU cycles. It is recommended that no other tasks be active, and that the ASYNC Communication application be written to reduce the CPU overhead as much as possible.

ASYNC Communication applications should have optimal data block sizes for DosRead or DosWrite requests to the physical ASYNC device driver. In general, as block size increases, the overhead related to making those requests decreases. However, these applications may need more time to handle large data chunks for CRC checking, or saving, the data into a file as this takes CPU cycles away from the physical ASYNC device driver.

- Prevention of the physical ASYNC device driver receive buffer overflow. Appropriate communication protocols may be required for this.
- Applications should not use the Error or Break Replacement operations. If these options are taken, the physical ASYNC device driver runs in Data/Status mode, and cannot take advantage of 80386 machine instructions, such as REP MOVED. The physical device driver then has to check each status byte, which slows down the performance.

This function is extended from "Function 41H – Set Bit Rate" on page 18-8.

* Trademark of the IBM Corporation

Function 44H – Transmit Byte Immediate

This function transmits byte immediate.

Parameter Packet Format

Field	Length
Character to be Transmitted	BYTE

Data Packet Format: None. Packet pointer must be NULL.

Returns: If the call is made with an invalid Data Packet pointer, or if there is already another character waiting to be transmitted immediately due to a previous Function 44H request that has not been fulfilled, then a *general failure* error is reported and this request is ignored. A Transmit Immediate request is considered fulfilled when the character is given to the transmit hardware.

Remarks: If a general failure is not returned, the physical device driver immediately transmits the byte contained in the Parameter Packet subject to the following conditions:

- If there is data currently in the transmit queue being transmitted, or waiting to be transmitted, the character to be transmitted immediately is placed at the logical front of the physical device driver transmit queue (not considered *in* the transmit queue), so that it is the next character to be given to the transmit hardware. If Automatic Receive Flow Control is enabled, then a XON or XOFF character can be placed ahead of the character to be transmitted immediately.
- This request always completes immediately (before the character is actually transmitted), even if the character might not be immediately transmitted. If there already is one character waiting to transmit immediately due to a previous request, then a general failure error is returned. The application must make this request again after there is no character (waiting to transmit immediately) in the device driver transmit queue. "Function 64H – Query COM Status" on page 18-42 can be used to determine whether a character is currently waiting to be transmitted immediately.
- The physical device driver does not immediately transmit the character (waiting to transmit immediately), if the physical device driver is not transmitting characters due to modem control signal output handshaking (see "Function 53H – Set DCB Parameters" on page 18-21), or if the physical device driver is currently transmitting a break.
- If the physical device driver is not transmitting characters due to Automatic Transmit or Receive Flow Control (XON/XOFF) being enabled, or due to being asked to operate as if an XOFF character had been received (see "Function 47H – Behave as if XOFF Received" on page 18-18), then the physical device driver still transmits a character that is waiting to be transmitted immediately due to this request.

Warning: An application, which requests the device driver to transmit a character immediately when Automatic Transmit or Receive Flow Control is enabled, can cause unexpected results to happen to the communications line flow control protocol.

- This is generally used to manually send XON and XOFF characters.
- The character waiting to be transmitted immediately is not considered part of the physical device driver transmit queue, and is not flushed due to a flush request. XON/XOFF characters that are automatically transmitted due to Automatic Receive Flow Control might be placed ahead of the character waiting to be transmitted immediately. Applications should not have dependencies on this ordering.

- If the serial port controller fully supports Extended Hardware Buffering capabilities, and the physical device driver is set with Extended Hardware Buffering enabled, then calling this function results in temporarily setting the Transmit Buffer Load Count to 1 (to load the transmit immediate byte). If the physical device driver conditions allow data to be transmitted, then the byte is sent, and the device driver then resumes operations with the previously prevailing Transmit Buffer Load Count (as determined by Function 53H).

Function 45H – Set Break OFF

This function sets the break off.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
COM Error WORD (COMERR)	WORD

COM Error WORD The physical device driver returns this information, if a general failure error is not reported. See “Function 6DH – Query COM Error” on page 18-48, for COMERR definition. The COM device error information is not cleared by this action.

Returns: If the call is made with an invalid Parameter Packet pointer and a *general failure* error is reported, this function is not performed and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, then the physical device driver stops generating a break signal. It is not considered an error, if the physical device driver is not generating a break signal. The physical device driver then resumes transmitting characters, taking into account all the other reasons why characters might be transmitted.

Function 46H – Set Modem Control Signals

This function sets the modem control signals.

Parameter Packet Format

Field	Length
Modem Control Signals ON Mask	BYTE
Modem Control Signals OFF Mask	BYTE

Modem Control Signals The physical device driver sets the modem control signals as defined in this field. Bit 0 is DTR; bit 1 is RTS. If any other bits are set/reset by the masks, then a general failure error results. The OFF mask contains a mask (with all bits equal to 0) to turn off. The ON mask contains a mask (with all bits equal to 1) to turn on. If the Parameter Packet shows turn off, and on on the same bit, the bit will be turned on.

For example:

Mask ON	Mask OFF	Description
01H	FFH	Set DTR
00H	FEH	Clear DTR
02H	FFH	Set RTS
00H	FDH	Clear RTS
03H	FFH	Set DTR and RTS
00H	FCH	Clear DTR and RTS.

If DTR Control mode input handshaking, RTS Control mode input handshaking, or *toggling on transmit* is set, then this request cannot try to change the state of the modem control signals, which is being used for input handshaking, or toggling on transmit. If the request tries to modify a modem control signal that is being used for input handshaking, or toggling on transmit, a *general failure* error results.

Data Packet Format

Field	Length
COM Error WORD (COMERR)	WORD

COM Error WORD The physical device driver returns this information, if a general failure error is not returned to the application. See the "Function 6DH – Query COM Error" on page 18-48 for COMERR definition. The COM device error information is not cleared by this action. At device driver initialization, the device driver turns off DTR and RTS, for the COM devices that it owns.

When the COM device is not already open (from a previous Open without a Close), the OPEN request packet causes DTR and RTS to be set according to the DTR Control mode and the RTS Control mode. See Note 1 of "Function 53H – Set DCB Parameters" on page 18-21.

Note: If the port will not be open after processing a CLOSE request packet (Last Level Close), DTR and RTS are turned off by the device driver. This occurs after the transmit hardware has completely transmitted (at the physical RS232 interface) all the data that it has been given by the physical device driver and time for at least 10 additional character transmissions has elapsed.

Returns: If the call is made with invalid Parameter Packet values and a general failure error is reported, then the modem control signals are not changed, and the Data Packet information returned to the application is *undefined*.

Function 47H – Behave as if XOFF Received

This function behaves as if XOFF received (stops transmitting).

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format: None. Packet pointer must be NULL.

Returns: If the call is made with invalid Parameter or Data Packet pointers, a *general failure* error is reported and this request is not performed by the physical device driver.

Remarks: If a general failure error is not returned by the physical device driver, this function causes data transmission to be stopped by preventing the physical device driver from sending additional data to the transmit hardware.

If Automatic Transmit Flow Control is enabled, this request causes the physical device driver to behave as if it had received the XOFF character. Transmission can be resumed, when an XON is received by the physical device driver after a “Function 48H – Behave as if XON Received” request is received, or when the physical device driver is told to disable Automatic Transmit Flow Control (if it was enabled).

If Automatic Transmit Flow Control is disabled, then the Function 48H request is required for transmission to be resumed. If, after this request is received, the physical device driver is told to enable Automatic Transmit Flow Control, then transmission is still disabled, but can be re-enabled. See Note 2 of “Function 53H – Set DCB Parameters” on page 18-21.

See “Function 64H – Query COM Status” on page 18-42 for other reasons why transmission might be disabled.

Function 48H – Behave as if XON Received

This function behaves as if XON received (starts transmitting).

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format: None. Packet pointer must be NULL.

Returns: If the call is made with invalid Parameter or Data Packet pointers, a *general failure* error is reported and this request is not performed by the physical device driver.

Remarks: If a general failure error is not returned by the physical device driver, this function allows data transmission to be resumed by the device driver if the data transmission was stopped due to a “Function 47H – Behave as if XOFF Received” request, or due to receiving an XOFF character while the physical device driver is in Automatic Transmit Flow Control mode.

See “Function 64H – Query COM Status” on page 18-42 for other reasons why transmission might be disabled.

Function 4BH – Set Break ON

This function sets break on.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
COM Error WORD (COMERR)	WORD

COM Error WORD The physical device driver returns this information, if a general failure error is not reported. See “Function 6DH – Query COM Error” on page 18-48, for COMERR definition. The COM device error information is not cleared by this action.

If, after processing this Close request, the port will not be open (from another Open without a Close), a CLOSE request packet causes break to be turned off.

Returns: If the call is made with an invalid Parameter Packet pointer and a *general failure error* is reported, this function is not performed and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver generates the break signal immediately. It is not considered an error, if the physical device driver is already generating a break signal. The device driver does not wait for the transmit hardware to become empty. Note that more data is not given to the transmit hardware until the break is turned off. The break signal is always transmitted, regardless of whether or not the physical device driver is transmitting characters for other reasons.

Function 53H – Set DCB Parameters

This function sets the Device Control Block (DCB) information.

Parameter Packet Format

Field	Length
Write Timeout	WORD
Read Timeout	WORD
Flags1	BYTE
Flags2	BYTE
Flags3	BYTE
Error Replacement Character	BYTE
Break Replacement Character	BYTE
XON Character	BYTE
XOFF Character	BYTE

Write Timeout Specifies the time period used for Write Timeout processing. The value is in units of .01 seconds based on 0, (where 0=.01 seconds). See Note 8.

Read Timeout Specifies the time period used for Read Timeout processing. The value is in units of .01 seconds based on 0, (where 0=.01 seconds). See Note 9.

Flags1 Has the following bits:

Bits 0-1 DTR Control Mode. See Note 1.

Bit 1	Bit 0	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Invalid input, resulting in a general failure error.

Bit 2 Reserved. Set to 0.

Bit 3 Enable output handshaking, using CTS. See Note 3.

Bit 4 Enable output handshaking, using DSR. See Note 3.

Bit 5 Enable output handshaking, using DCD. See Note 3.

Bit 6 Enable input sensitivity, using DSR. See Note 4.

Bit 7 Reserved. Set to 0.

Flags2 Has the following bits:

Bit 0 Enable Automatic Transmit Control Flow (XON/XOFF). See Note 2.

Bit 1 Enable Automatic Transmit Control Flow (XON/XOFF). See Note 2.

Bit 2 Enable error replacement character. See Note 5.

Bit 3 Enable null stripping. See Note 6.

Bit 4 Enable break replacement character. See Note 7.

Bit 5 Automatic Receive Flow Control:

0 = Normal

1 = Full Duplex.

Bits 6-7 RTS Control Mode. See Note 1.

Bit 7	Bit 6	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Toggling on transmit.

Flags3

Has the following bits:

Bit 0 Enable Write Infinite Timeout processing. See Note 8.

Bits 1-2 Enable Read Timeout processing. See Note 9.

Bit 2	Bit 1	Description
0	0	Invalid input. Results in a general failure error.
0	1	Normal Read Timeout processing
1	0	Wait-For-Something Read Timeout processing
1	1	No-Wait Read Timeout processing.

Bits 3-4 Extended Hardware Buffering. See Note 10.

Bit 4	Bit 3	Description
0	0	Not supported; ignored. No change to FIFO state.
0	1	Disabled
1	0	Enabled
1	1	Automatic Protocol Override.

Bits 5-6 Received trigger level. See Note 11.

Bit 6	Bit 5	Description
0	0	1 character
0	1	4 characters
1	0	8 characters
1	1	14 characters.

Note: The trigger level must be set to 1 character when Enhanced mode is on.

Bit 7 Transmit Buffer Load Count. See Note 12.

0 = 1 character
1 = 16 characters.

Note: When Extended Hardware Buffering is disabled, this bit is set by the device driver.

Error Replacement Any byte value in the range 00H – FFH. See Note 5.

Break Replacement Any byte value in the range 00H – FFH. See Note 7.

XON Character Any byte value in the range 00H – FFH. See Note 2.

XOFF Character Any byte value in the range 00H – FFH. See Note 2.

Data Packet Format: None. Packet pointer must be NULL.

Returns: If the call is made with invalid Parameter Packet values or an invalid Data Packet pointer, a *general failure* error is reported and none of the Device Control Block (DCB) characteristics of the physical device driver for this COM device are changed.

Remarks: If a general failure error is not returned, then the actions described below are taken by the physical device driver. Notice that all reserved bit fields that are specified as *Set to 0* must be set to zero on entry to the physical device driver, or a general failure error results. The same applies to the NULL Data Packet pointer.

The general DCB parameter access, “Function 53H – Set DCB Parameters” on page 18-21, and “Function 73H – Query DCB Parameters” on page 18-50 are used:

- For Automatic Transmit Flow Control (start/stop transmit, when XON/XOFF character received)
- For Automatic Receive Flow Control (transmit XON/XOFF, when receive buffer fills or empties)
- To determine XON/XOFF characters
- For DTR Control mode (enable/disable/input handshaking)
- For RTS Control mode (enable/disable/input handshaking/toggling on transmit)
- For output handshaking using CTS/DSR/DCD (control signal determines when to transmit)
- For input sensitivity using DSR (reception of data controlled by DSR)
- For error replacement character and processing
- For break replacement character and processing
- For null stripping
- To Receive or Transmit Timeout processing.

To maintain upward compatibility, the application should call “Function 73H – Query DCB Parameters,” before Function 53H is used. This allows the *reserved* bits to be set correctly in a future release of the physical device driver. By doing the return first, the application maintains the state of the physical device driver for a mode that the application is not aware of.

Note 1: Control of DTR and RTS. The physical device driver allows the caller to automatically control the setting of Data Terminal Ready (DTR) and Request To Send (RTS) through the RTS Control mode and the DTR Control mode settings of “Function 53H – Set DCB Parameters.” The application can also request manual control over these modem control signals. The ways in which these signals are controllable are as follows:

Set RTS Control Mode to Toggling on Transmit.: If the Flags2 bits 7, 6 are set to 1, 1, then the physical device driver is in the automatic control mode of RTS. When the physical device driver is initialized, the RTS Control mode is enabled, therefore, initially the device driver is not in the automatic control mode of RTS. Notice that this mode of operation of the physical device driver should only be enabled when the system is attached to devices, which do not present data to the system receive hardware when RTS is *on*. In this mode, the device driver:

- Always turns on RTS, if a break is being transmitted.
- Once data is in the transmit hardware buffer, does not turn RTS *off* until the transmit hardware has emptied its buffers.
- Turns on RTS, if not already *on*, when there is data in the physical device driver transmit queue, or when there is an outstanding WRITE request packet, and:
 - The physical device driver is allowed to transmit, even if Automatic Transmit/Receive Flow Control (XON/XOFF) is enabled. The physical device driver needs to turn on RTS momentarily to transmit a character immediate, if not normally allowed to transmit due to Automatic Transmit/Receive Flow Control.
 - The physical device driver is allowed to transmit because it was not told to behave as if an XOFF had been received (Function 47H). The physical device driver needs to turn on RTS momentarily to transmit a character immediate, if not normally transmitting due to XOFF flow control considerations. Also the physical device driver needs to turn on RTS momentarily to transmit an XON or XOFF due to Automatic Receive Flow Control, if not normally transmitting due to XOFF flow control considerations.

- Turns off RTS, if not already *off*, when one of the following conditions are TRUE:
 - No data in the physical device driver transmit queue (and no data in Write requests in progress), no queued Write requests, and the transmit hardware has physically transmitted (at the physical RS232 interface) all the data that it has been given.
 - The physical device driver is not allowed to transmit, due to Automatic Transmit/Receive Flow Control (XON/XOFF) being enabled, or due to being asked to behave as if an XOFF had been received (Function 47H). The physical device driver needs to turn on RTS to transmit a character immediate or XON/XOFF, due to Automatic Receive Flow Control (XON/XOFF). RTS is never turned *off* until the transmit hardware has physically transmitted all the data that it has been given.
- When this function is enabled, the physical device driver controls RTS, as determined by the above description.
- If this function is disabled by choosing a new RTS Control mode, then RTS is controlled by the new RTS Control mode that is inherently chosen, when this RTS Control mode is disabled.
- The device driver does not examine any other modem control signals before it turns RTS *off* or *on*.

An OPEN request packet does not cause the physical device driver to change its RTS Control mode. The physical device driver maintains the state of this mode of operation across OPEN request packets. When the physical device driver is in the RTS Control mode, *toggle on transmit*, then it does not allow the application to control RTS by using "Function 46H – Set Modem Control Signals" on page 18-16.

Set DTR and RTS Control Mode to Input Handshaking: When the Flags1 bits 1, 0 are set to 1, 0, the DTR Control mode is set to input handshaking. Setting bits 7, 6 of Flags2 to 1, 0, sets the RTS Control mode to input handshaking. When the physical device driver is initialized, the RTS and DTR Control modes are enabled; so initially the device driver is not in the automatic control mode of RTS and DTR. Notice that this mode of operation of the physical device driver is only set when there is the possibility of a physical device driver receive queue overrun, and the system is attached to data terminal equipment that stops transmitting data when the appropriate modem control signals are turned *off*.

Because the Input Handshaking mode can be set for RTS, DTR, or both, the DTR and RTS Control modes are processed independently. In Input Handshaking mode, the physical device driver:

- Turns the appropriate modem control signals *on* when the device driver receive queue is less than half-full
- Turns the appropriate modem control signals *off* when the device driver receive queue is almost full
- Does not monitor the value of the appropriate modem control signals (when this mode is first set and the queue size is between half-full and almost full)
- When this function is enabled, determines the correct value of the modem control signals, and controls them accordingly

If this function is disabled (by choosing a new RTS or DTR Control mode), RTS or DTR is controlled by the new RTS or DTR Control mode that is inherently chosen when this RTS or DTR Control mode is disabled

- Does not examine any other modem control signals before controlling DTR or RTS.

An OPEN request packet does not cause the physical device driver to change its RTS and DTR Control modes. The driver maintains the state of these modes of operation across OPEN request packets. When the physical device driver is in the RTS Control mode input handshaking, it does not allow the application to control RTS through Function 46H. When the physical device driver is in the DTR Control mode input handshaking, it does not allow the application to control DTR through Function 46H.

Set DTR and RTS Control Mode to Enable or Disable: OPEN processing has the following settings:

- Flags1, bits 1, 0 are set to 0, 0. DTR Control mode is disabled.
- Flags1, bits 1, 0 are set to 0, 1. DTR Control mode is enabled.
- Flags2, bits 7, 6 are set to 0, 0. RTS Control mode is disabled.

- Flags2, bits 7, 6 are set to 0, 1. RTS Control mode is enabled.

When the physical device driver is initialized, the RTS and DTR Control modes enable, but the value of the modem control signals is *off* until the port gets an OPEN request packet. An OPEN request packet does not cause the physical device driver to change its RTS and DTR Control modes. The driver maintains the state of these modes of operation across OPEN request packets.

Because Enable or Disable modes can be set for either RTS, DTR, or both, the DTR and RTS Control modes are processed independently. If the RTS Control mode is disabled when the physical device driver receives an OPEN request packet, and the device is not already open (from a First Level Open), the RTS modem control signal is kept (turned) *off* during the OPEN processing. If the RTS Control mode is enabled when the physical device driver receives an OPEN request packet, and the device is not already open, the RTS modem control signal is turned *on* during the OPEN processing.

If the RTS Control mode is set to disable, and the previous mode was not disable, the RTS modem control signal is turned *off*. If the RTS Control mode is set to disable, and the previous mode was also disable, this IOCTL has no effect on the RTS modem control signal.

If the RTS Control mode is set to enable, and the previous mode was not enable, the RTS modem control signal is turned *on*. If the RTS Control mode is set to enable, and the previous mode was also enable, this IOCTL has no effect on the RTS modem control signal.

The following tables summarize the above information:

From Control Mode	To Control Mode	Effect on Control Signal
Disable	Disable	None
Disable	Enable	Turn ON
Disable	Input handshaking	Modem control signal controlled automatically
Enable	Disable	Turn OFF
Enable	Enable	None
Enable	Input handshaking	Modem control signal controlled automatically
Input handshaking	Disable	Turn OFF
Input handshaking	Enable	Turn ON
Input handshaking	Input handshaking	Modem control signal controlled automatically

DTR and RTS Control Modes

From Control Mode	To Control Mode	Effect on Control Signal
Disable	Toggle on transmit	Modem control signal controlled automatically
Enable	Toggle on transmit	Modem control signal controlled automatically
Input handshaking	Disable	Modem control signal controlled automatically
Toggle on transmit	Disable	Turn OFF
Toggle on transmit	Enable	Turn ON
Toggle on transmit	Input handshaking	Modem control signal controlled automatically
Toggle on transmit	Toggle on transmit	Modem control signal controlled automatically

RTS Control Mode Only

Because the initial Control mode of the physical device driver is enabled for RTS and DTR, both modem control signals are turned *on* when the port is first opened. If the physical device driver receives an OPEN

request packet, and the device is already open, the physical device driver does not alter the value of the RTS and DTR modem control signals, regardless of the control mode.

The application can explicitly turn DTR or RTS *on* or *off* with Function 46H. If the Control mode of RTS is not enable, or disable, the application cannot control RTS with Function 46H, because the physical device driver is controlling the signal automatically using toggling on transmit, or input handshaking. If the control mode of DTR is not enable, or disable, the application cannot control DTR with Function 46H, because the device drive is controlling the signal automatically using input handshaking.

In CLOSE processing, when the physical device driver receives a CLOSE request packet and the COM device is still open, the physical device driver does not change the values of DTR or RTS. If the port is not open (from a Last Level Close) after processing a Close request, then, at the end of CLOSE processing, RTS and DTR are turned *off* after waiting the appropriate amount of time.

Note 2: Automatic Flow Control (XON/XOFF). If bit 0 of Flags2 is set, the device driver is enabled for Automatic Transmit Flow Control. If bit 1 is set, the physical device driver is enabled for Automatic Receive Flow Control. When the physical device driver is initialized, these bits are reset, so the physical device driver is not enabled for automatic transmit or receive flow control.

An OPEN request packet does not cause the physical device driver to change the enabling or disabling state of Automatic Transmit/Receive Flow Control. An OPEN request packet, when the COM device is not already open, causes the physical device driver to act as if it has not received an XOFF, if Automatic Transmit Flow Control is enabled. This OPEN also causes the device driver to act as if it has not transmitted an XOFF, if Automatic Receive Flow Control is enabled.

Automatic Transmit Flow Control (XON/XOFF): When XON and XOFF flow control during transmission is enabled, the device driver stops sending data to the transmit hardware when an XOFF is received, and resumes sending data to the transmit hardware when an XON is received.

Note: Although the physical device driver can transmit XON or XOFF, there are reasons why the it might not be able to transmit an XON or XOFF, such as transmitting break, or invalid output handshaking on modem control signals. Also, there are reasons, not related to Automatic Transmit/Receive Flow Control, why the physical device driver might not be able to resume transmitting data. See “Function 64H – Query COM Status” on page 18-42.

The physical device driver transmits characters, due to the transmit immediate request (see “Function 44H – Transmit Byte Immediate” on page 18-13). The physical device driver also transmits XON and XOFF, due to Automatic Receive Flow Control. When the physical device driver is in this mode, it does not pass received XON and XOFF characters to the application. Instead, the physical device driver acts upon receiving those characters and discards them.

The physical device driver can transmit additional characters before it recognizes an XOFF character that it has not read but is in the receive buffer of the hardware. The extent of this scenario is minimized, but the combined transmit/receive Advanced BIOS request block is still used on systems that support ABIOS. If the system is relatively slow in responding to interrupts, compared to the current bit rate, receive buffer overruns might not occur, but the physical device driver might seem slow in responding to an XOFF character.

If Automatic Transmit Flow Control is disabled (after being currently enabled), and transmission was not occurring, due to an XOFF received or at the request of “Function 47H – Behave as if XOFF Received,” then transmission is resumed. It is the responsibility of the application not to fully close the port in a way that causes the physical device driver to illegally transmit characters when the port is re-opened after being fully closed (First Level Open).

Output handshaking, using modem control signals, is one way that the physical device driver can be told to stop transmitting. See Note 3.

Automatic Receive Flow Control (XON/XOFF): When XON and XOFF flow control during receive is enabled, the device driver transmits an XOFF when its receive queue gets almost full, and an XON when its receive queue is about half-full.

When the COM device driver is in Normal mode of Automatic Receive Flow Control after the XOFF is sent, it sends no characters until it sends an XON (due to the reduction of the amount of data in its receive queue). This is to accommodate those systems that interpret the first character received, after an XOFF, as an XON, regardless of what the character actually is. The physical device driver transmits characters due to the transmit immediate request (Function 44H).

When the COM device driver is in the Full Duplex mode of Automatic Receive Flow Control after the XOFF is sent, it continues to send characters even though the receive queue remains almost full. This mode of the physical device driver is set by using bit 5 of Flags2.

The physical device driver is not able to transmit an XOFF or XON if it is transmitting a break. It is not able to automatically transmit an XOFF or XON, if it is enabled for output handshaking with certain modem control signals, and those modem control signals are not *on*. Notice that *this could cause a deadlock, if the physical device driver tries to transmit an XON and cannot*. The physical device driver will continue to transmit an XOFF or XON when transmit conditions permit, assuming the receive queue conditions still warrant it.

The physical device driver does not monitor characters being transmitted by WRITE request packets to see if any of them are XON or XOFF. It also does not monitor characters transmitted immediately. For example, the device driver does not stop transmitting characters, if the application causes it to explicitly transmit an XOFF.

If Automatic Receive Flow Control is enabled (after being currently disabled), the physical device driver immediately checks the receive queue level to see if an XOFF needs to be transmitted. An XON is never transmitted immediately, when this function is enabled. The physical device driver automatically transmits an XON character *only* after it has automatically transmitted an XOFF character.

If Automatic Receive Flow Control is disabled (after being currently enabled), and transmission was not occurring due to automatic transmission of an XOFF character (in the Normal mode of Automatic Receive Flow Control only), the physical device driver transmits an XON, and transmission is resumed, if possible. Notice that transmission might not be taking place for other reasons (see "Function 64H – Query COM Status" on page 18-42).

If Normal Automatic Receive Flow Control is currently enabled, and transmission is not occurring due to automatic transmission of an XOFF character, then when the physical device driver is called to enable Full-Duplex Automatic Receive Flow Control, the physical device driver immediately begins sending data regardless of the state of the receive queue. An XON character is sent only when the receive queue becomes half-full.

If the physical device driver has previously automatically transmitted an XOFF and a CLOSE request packet is received, and, after processing this Close request the port will not be open, the physical device driver automatically transmits an XON, if possible. It is the responsibility of the application not to fully close the port in a way that causes the physical device driver to illegally transmit characters, or the communications link to enter a deadlock state when the port is re-opened.

Input handshaking using modem control signals is one way that the device driver can tell another device to stop transmitting. See Note 1.

XON and XOFF Characters: The value of these bytes in the device control block determine the value of the XON and XOFF characters used for automatic transmit and receive flow control. When the XON and XOFF characters are referred to in the Category 1 IOctls, the reference is to the value of the XON and XOFF characters as determined by this IOctl.

When the physical device driver is first initialized, the XON character is 11H, and the XOFF character is 13H. An OPEN request packet, when the COM device is not already open, causes the XON character to be set to 11H, and the XOFF character to be set to 13H. If the XON and XOFF characters are set equal with this IOctl, the results are *undefined*.

Note 3: Output Handshaking Using CTS, DSR, and DCD. Bits 3, 4, and 5 of Flags1 control Output Handshaking Using CTS, DSR, and DCD respectively. If the bit is set, output handshaking for the appropriate modem control signal is enabled. Output Handshaking mode can be enabled for any combination of CTS, DSR, or DCD because bit 3, 4, and 5 of Flags1 can be set independently. The data is not transmitted unless all the lines enabled for output handshaking are up.

When the physical device driver is initialized, bits 3 and 4 of Flags1 are set, and bit 5 of Flags1 is reset. Therefore, initially the device driver is enabled for output handshaking using CTS and DSR, but disabled for output handshaking using DCD. Except for attachment to special devices or special cables, output handshaking using DCD should not be enabled.

Disabling Output Handshaking Using CTS or DSR causes unexpected results when the system is attached to data terminal devices, or to data communications devices that toggle CTS or DSR. If the device driver is enabled for this mode of operation, it is affected in the following manner if the appropriate modem signals are *off*:

- The physical device driver is unable to move data from the physical device driver transmit queue to the transmit hardware.
- The physical device driver is unable to transmit a character immediately (Function 44H), so that the character is “remembered” by the device driver.
- The physical device driver is unable to automatically transmit XONs and XOFFs. (The physical device driver might try to transmit XONs and XOFFs as a result of Automatic Receive Flow Control enabled.)
- The physical device driver generates a break immediately, if requested.
- The value of CTS, DSR, and DCD does not affect how the physical device driver controls RTS and DTR.

An OPEN request packet does not cause the physical device driver to change the value of bits 3, 4, and 5 of Flags1. It maintains the state of this mode of operation across OPEN request packets.

On devices with a transmit holding register and transmit shift register, the transmit holding register is always given another character to transmit when it empties (even though a character can still be in the transmit shift register), unless the physical device driver determines that it is no longer allowed to transmit. The physical device driver always attempts to detect a change in the modem status signals (CTS, DSR, DCD) before transmitting more data.

Note 4: Input Sensitivity Using DSR. Bit 6 of Flags1 controls input sensitivity using DSR. If the bit is set, input sensitivity using DSR is enabled. When the physical device driver is initialized, bit 6 of Flags1 is set; so initially the physical device driver is enabled for input sensitivity using DSR.

Note: Disabling input sensitivity using DSR causes unexpected results when the system is attached to data terminal devices, or to data communications devices that toggle DSR when they generate spurious data that the system should not receive.

If the physical device driver is enabled for this mode of operation, it throws away all data input from the receive hardware when DSR is *off*. If the physical device driver processes a change in the DSR modem control signal from *on* to *off*, or from *off* to *on* at the same time that it inputs a character from the receive hardware, then it still accepts the last characters. This can cause the physical device driver to attempt to process invalid data for one service period of the receive hardware. Therefore, it is required that the change in the modem control signal be processed *before* the physical device driver attempts to receive data from the receive hardware (see Note 3), or that the received data be saved until a change in modem status (during the same hardware service instance) is determined.

An OPEN request packet does not cause the physical device driver to change the value of bit 6 of Flags1. The physical device driver maintains the state of this mode of operation across OPEN request packets.

Note 5: Error Replacement Character. The Flags2 bit 2 controls the enabling of error replacement character processing. If set, processing is enabled. When the physical device driver is initialized, this bit is reset, so initially the physical device driver is not enabled for the error replacement character. An OPEN request packet, when the COM device is not already open, causes this bit to be reset, disabling error replacement character processing.

When the physical device driver is initialized, the error replacement character is 00H. An OPEN request packet, when the COM device is not already open, causes the error replacement character to be set back to 00H.

If error replacement character processing is disabled, the following applies:

- If a parity or framing error occurs, and the character with the error is available in the receive hardware buffer, it is placed in the device driver receive queue.
- If a hardware or receive queue overrun occurs, nothing is placed into the receive queue to designate an overrun.

If error replacement character processing is enabled, the following applies:

- If a parity or framing error occurs, the error replacement character is placed into the physical device driver receive queue, instead of the character in the receive hardware buffer, if it is available.
- If a hardware buffer overrun occurs, the error replacement character is placed into the physical device driver receive queue to mark the position that a receive overrun occurred. If valid data is in the receive hardware buffer, it is placed into the device driver receive queue. The processing of the valid data takes place after the hardware buffer overrun condition is recorded in the device driver receive queue.
- If the user requests the error replacement character option while a receive or transmit operation is in progress, there is a chance of loss in DMA mode. The user, and the transmitting system, should stop transmitting. The user should also check the device driver receive and transmit queues to ensure that they are empty, before requesting the error replacement character option.
- If a physical device driver receive queue overrun occurs, the last character in the receive queue is replaced with the error replacement character. This allows the application to know the position where the error occurred. This error replacement, if enabled, always takes precedence over an error replacement or break replacement event that occurred for the same character.

Regardless of whether error replacement character processing is enabled, null stripping and checking for XON/XOFF characters does not occur if the character had an error. This IOctl can be used to change the error replacement character by changing the byte representing the error replacement character.

Note 6: Null Stripping. Bit 3 in Flags2 controls the enabling of null stripping processing. If set, null stripping processing is enabled. When the physical device driver is initialized, this bit is reset, so initially the physical device driver is not enabled for null stripping. An OPEN request packet, when the COM device is not already open, causes this bit to be reset, disabling null stripping.

If the physical device driver is enabled for null stripping when characters are read in from the receive hardware, any *non-error* or *non-break* characters with a value of 00H are discarded, not checked (even if the XON or XOFF character has been set to 00H), and not placed into the physical device driver receive queue.

Note: Simultaneously setting the XON or XOFF character to 00H, enabling Automatic Transmit Flow Control, and enabling null stripping can cause unexpected results, but is not considered an error condition by the physical device driver error checking logic.

Note 7: Break Replacement Character. Bit 4 in Flags2 controls the enabling of break replacement character processing. If set, processing is enabled. When the physical device driver is initialized, this bit is reset, so initially the physical device driver is not enabled for the break replacement character.

An OPEN request packet, when the COM device is not already open, causes this bit to be reset, disabling break replacement character processing. When the physical device driver is initialized, the break replacement character is 00H. An OPEN request packet, when the COM device is not already open, causes the break replacement character to be reset back to 00H.

If break replacement character processing is disabled, the device driver does not place any character in the physical device driver receive queue when it detects a break condition on the line. A detected break condition has no effect on XON/XOFF detection. If break replacement character processing is enabled, if the physical device driver detects a break condition, it places the break replacement character in the device driver receive queue.

Note: If the user requests the break replacement character option while a receive or transmit operation is in progress, there is a chance of loss in DMA mode. The user, and the transmitting system, should stop transmitting. The user should also check the device driver receive and transmit queues to ensure that they are empty, before requesting the break replacement character option.

If break replacement character processing is enabled, null stripping and checking for XON/XOFF characters do not operate on the break replacement character. This IOCTL can be used to change the break replacement character by changing the byte representing the break replacement character.

If a parity or framing error is generated due to the reception of a break, error replacement processing is not done (except for the *overrun* condition), instead, break replacement processing is done.

Note 8: Write Timeout. Bit 0 in Flags3 controls the characteristics of Write Timeout processing. If the bit is 0, Write Timeout processing uses the value in the Write Timeout WORD in the device control block. If the bit is 1, Write Timeout processing is infinite timeout.

The value in the Write Timeout WORD is in .01 second units, based on 0 (where, 0 = .01 seconds). The physical device driver is considered to be doing Normal Write Timeout processing when the Write Timeout WORD is used.

During Normal Write Timeout processing, if the physical device driver does not give any data to the transmit hardware from the transmit queue within the period of time specified by the Write Timeout WORD, the request is completed. The accuracy of the timeout period can be determined by the request packet, which is blocked in the device driver, and how long it takes for the thread to be dispatched once it is made ready by the expiration of the timeout period. The accuracy of the timeout period can also be determined by the accuracy of the device driver timer tick processing. If any data had been given to the transmit hardware in that timeout period, the specified period of time is waited for again, to see if any more data has been transmitted.

If the timeout period is changed by this IOCTL to Infinite Timeout, the new time can take effect immediately, or can take effect after the next character is written.

During Write Infinite Timeout processing, the request does not complete until all the data from the request has been given to the transmit hardware. The thread of the Write request does not return to the system until the request completes. The physical device driver checks to see if an IOCTL has changed the Write Timeout processing characteristics at least every minute. This can occur almost immediately (accuracy can be determined by the request packet, which is blocked, or by device driver timer ticks), and insures that the device driver periodically checks to see if Write Infinite Timeout processing has been changed to Normal Write Timeout processing. The Write Timeout characteristics can be changed in the middle of the processing of a Write request, and the new timeout attribute is guaranteed to eventually take effect. When the physical device driver initializes, Normal Write Timeout processing is in effect.

When the physical device driver receives an OPEN request packet for the port, and the port is not already open, the value in the Write Timeout WORD is set to one minute. The current Write Timeout processing characteristics (normal or infinite) are not affected.

Note 9: Read Timeout. Bits 2, 1 of Flags3, control the Read Timeout processing characteristics of the physical device driver. The three possible types of Read Timeout processing are:

Normal	Bits 2, 1 = 0, 1
Wait-For-Something	Bits 2, 1 = 1, 0
No-Wait	Bits 2, 1 = 1, 1.

The value in the Read Timeout WORD is in .01 second units, based on 0 (where 0 = .01 seconds). The physical device driver uses the value in the Read Timeout WORD for Normal and Wait-For-Something Read Timeout processing. The accuracy of the time interval can be determined by the request, which is blocked in the physical device driver, or by the device driver timer ticks.

If the physical device driver is doing Normal Read Timeout processing, the device driver waits as long as the value in the Read Timeout WORD indicates to wait. The request is completed after that interval of time elapses, if no more data has been received for the request. If any data is received by the physical device driver from the receive hardware for the request (including XON/XOFF characters), the specified period of time is waited on again for more data to arrive. However, in the following two cases, the current interval of time will continue to be waited on without starting to wait from the beginning of the interval again:

- If input sensitivity using DSR is enabled, and the value of the DSR modem control signal causes input data to be thrown away. See Note 4.
- If null stripping is enabled, and a null character is stripped. See Note 6.

If the physical device driver is doing No-Wait Read Timeout processing, it does not wait for any data to be available in the receive queue. When the physical device driver begins to try to move data from the receive queue to the request, the request completes. Whatever data is available in the receive queue at that time, is the amount of data that is moved to the request.

If the physical device driver is doing Wait-For-Something Read Timeout processing, the physical device driver processes the request initially as if it had No-Wait Timeout processing. If no data was available at the time the request would have completed due to No-Wait processing, the request is not completed. Instead, it waits for some data to be available before completing the request. However, the physical device driver does enter Normal Read Timeout processing for this request. Therefore, if no data is available after the Normal Timeout processing interval, then the request is completed anyway. The request never waits longer than it would have due to Normal Read Timeout processing.

The Read Timeout processing characteristics that apply to a given Read request are not determined until the physical device driver begins processing that request. At that time, a change to the Read Timeout processing characteristics of the physical device driver between Wait-For-Something and Normal Timeout processing might take effect for the current Read request being processed. If the timeout period is changed by this IOctI, the new timeout period might take effect immediately, or it might take effect after the next character is received from the receive hardware. When the physical device driver initializes, Normal Read Timeout processing is in effect.

When the physical device driver receives an OPEN request packet for the port, and the port is not already open, the value in the Read Timeout WORD is set to one minute, and Normal Read Timeout processing characteristics are put into effect.

Note 10: Extended Hardware Buffering. Refers to the capability of the COM port's serial controller device to buffer up to 16 characters in its internal hardware Receive and Transmit buffers. This buffering capability allows the device to relieve the operating system of the high overhead associated with servicing per-character receive and transmit hardware interrupts.

On COM ports with a serial controller device that does not support Extended Hardware Buffering, bits 3 and 4 of DCB Flags3 are always set to 0, indicating that the device does not support Extended Hardware Buffering. This value is always valid and ignored as input to Function 53H. If the device does not support Extended Hardware Buffering, and the application attempts to set any of the Flags3 bits 3–7, this IOCTL fails and a general failure error results.

Applications must first call "Function 73H – Query DCB Parameters" on page 18-50 to determine whether the device supports Extended Hardware Buffering *before* calling Function 53H.

When in conventional Programmed Input/Output (PIO) mode, or when the Enhanced mode is disabled through "Function 54H – Set Enhanced Mode Parameters," the physical ASYNC device driver defaults the setting of the Extended Hardware Buffering parameter to Automatic Protocol Override. This system default can be changed by manipulating bits 3 and 4 in the Flags3 parameter of Function 53H. An application or subsystem can manually control this feature by setting Extended Hardware Buffering enabled, and manipulating the Receive Trigger Level and Transmit Buffer Load Count parameters. An application or subsystem can also set the serial device to run in Character mode by setting Extended Hardware Buffering disabled. The three settings for this feature are described below:

Note: On a COM port with Enhanced UART, or compatibles, the system default mode of operations is the Enhanced mode where the minimum level of operation is Enhanced FIFO mode (the full capacity of the hardware FIFO buffer is automatically exploited and controlled by the device driver). When a DMA channel is available in the system at the moment of the user's Open request for a receive operation, or the user's Transmit request, the operation is performed in DMA mode, by default.

In either DMA or Enhanced FIFO mode, the settings for Flags3 in the Extended Hardware Buffering (including Automatic Protocol Control, Receiver Trigger Level, and Transmit Buffer Load Count) are ignored and have no effect. The user is allowed to disable the Enhanced mode through Function 54H. When the Enhanced mode is disabled, the device driver operates in conventional PIO mode, and does not exploit the hardware capabilities of the DMA or the advanced function features. Therefore, the hardware FIFO buffer operation depends on the Extended Hardware Buffering Flags3 bits to be set in a compatible manner.

Automatic Protocol Override (System Default): In any system configuration, where a COM port's serial controller correctly supports Extended Hardware Buffering, the physical device driver initializes that COM port to enable Automatic Protocol Override mode. This mode of the physical ASYNC device driver is defined with respect to the following device driver protocols:

- Output Handshaking Using CTS, DSR, DCD
- Automatic Transmit Flow Control
- Input Sensitivity Using DSR.

When any one or more of the above protocols is enabled, the Automatic Protocol Override feature causes the Extended Hardware Buffering to not fully exploit the maximum potential performance benefit of the serial controller. Depending on which protocols are enabled, the physical device driver automatically adjusts either, or both, of the Receive Trigger Level and Transmit Buffer Load Count. The following descriptions identify the changes that each of the above protocols cause when Automatic Protocol Override mode is active:

- Output Handshaking Using CTS, DSR (Default *on*)
- Output Handshaking Using DCD (Default *off*). When either of these handshaking protocols is enabled, the physical ASYNC device driver, under Automatic Protocol Override, sets the Transmit Buffer Load Count to 1. This means that it services transmit interrupts one character at a time. When these

protocols are disabled, the device driver fully exploits the Extended Hardware Buffering capabilities of the serial controller by transmitting 16 characters per interrupt. The Receive Trigger Level is not affected by these protocols in Automatic Protocol Override mode.

- **Automatic Transmit Flow Control (Default off).** When this protocol is enabled, the physical ASYNC device driver, under Automatic Protocol Override, sets the Receive Trigger Level and the Transmit Buffer Load Count to 1. This means that it services both receive and transmit interrupts one character at a time. When these protocols are disabled, the physical device driver fully exploits the Extended Hardware Buffering capabilities of the serial controller by transmitting 16 characters per interrupt, and setting the Receive Trigger Level to 8. (The physical device driver is given at least 8 characters on each Receive Data Available interrupt.)
- **Input Sensitivity Using DSR (Default on).** When this protocol is enabled, the Automatic Protocol Override feature of the ASYNC device driver sets the Receive Trigger Level to 1, by default, on the respective COM port, forcing the physical device driver to service all characters received one character at a time. The Transmit Buffer Load Count is not affected by this protocol.

With all of the above listed protocols, Automatic Protocol Override allows the serial controller to remain in its FIFO-mode setting, although it is not fully exploiting the potential performance benefit from the Extended Hardware Buffering capability. The extended Receive Hardware Buffering capability remains active so that receive hardware overrun errors are much less likely to occur.

When Automatic Protocol Override mode is enabled, setting the DCB Flags3 bits for manipulating Receive Trigger Level and Transmit Buffer Load Count (bits 5, 6, and 7) has no effect. Automatic Protocol Override fully overrides any manual settings the application or subsystem might attempt to set.

Note: Having any of the above protocols enabled can significantly alter system performance characteristics with respect to activity on an asynchronous communications port. This is particularly true where the COM port is running at a high bit rate (2400 bps or higher), or where multiple COM ports are performing asynchronous I/O. This is a factor to consider when configuring a system to perform multiple serial port I/O, or when developing an application that requires high speed asynchronous communications. Disabling the above protocols, or setting the Device Control Block parameters to Extended Hardware Buffering enabled, allows the physical device driver to fully exploit the serial controller's capability to its maximum benefit.

Extended Hardware Buffering Enabled: Setting this mode cancels the effects of Automatic Protocol Override and enables the user to fully use the Extended Hardware Buffering capabilities of the physical device driver. The Receive Trigger Level should be set to 8 (the physical device driver receives 8 characters per interrupt), and the Transmit Buffer Load Count should be set to 16 (the physical device driver transmits 16 characters per interrupt).

Setting this mode in conjunction with any of the following device driver protocols can alter the behavior of that protocol in a significant manner:

- Output Handshaking using CTS, DSR, DCD
- Automatic Transmit Flow Control
- Input Sensitivity using DSR.

For information on these protocols, refer to the description in "States of the ASYNC Device Driver" on page 8-8.

Extended Hardware Buffering Disabled: Setting this mode completely disables the Extended Hardware Buffering capabilities of the serial port controller. This mode places the serial port device into Character mode, that is, the physical device driver services both transmit and receive interrupts one character at a time. This effectively eliminates any special considerations that might have been necessary when performing asynchronous communications with Extended Hardware Buffering enabled, or with the Automatic Protocol Override mode active.

On any COM port, which is serviced by a serial controller that does not support FIFO mode, the physical device driver is always set to Extended Hardware Buffering disabled. Attempts to set Extended Hardware Buffering enabled, or to enable the Automatic Protocol Override mode, is ignored (no error returned). Whenever “Function 73H – Query DCB Parameters” is called, the bits in DCB Flags3 identify the true state of Extended Hardware Buffering on these devices as disable.

Note: Setting the physical device driver to run with Extended Hardware Buffering disabled can significantly alter the system performance characteristics, particularly with respect to asynchronous communications I/O throughput. This is especially true where the COM port is running at a high bit rate (2400 bps or higher), or where multiple COM ports are performing asynchronous I/O.

Note 11: Receive Trigger Level. Represents the number of characters that must be received by a serial port controller that supports Extended Hardware Buffering before that device generates a receive hardware interrupt. For example, with Extended Hardware Buffering enabled, assume the Receive Trigger Level is set to 8 characters. This means that a receive hardware interrupt occurs once for every 8 characters received at that COM port. On COM ports with a serial controller device that does not support Extended Hardware Buffering, bits 5 and 6 of DCB Flags3 are always set to 0, indicating that the device is in Character mode (Receive Trigger Level = 1).

The default state for bits 5 and 6 of DCB Flags3 is zero, indicating that the Automatic Protocol Override mode is active and that other device driver protocols are enabled, which force Automatic Protocol Override to keep Receive Trigger Level set to 1.

When the physical device driver is operating in Automatic Protocol Override mode, setting the Receive Trigger Level has no effect. This parameter setting is completely ignored unless DCB Flags3 bits 3 and 4 are set to Extended Hardware Buffering enabled.

Note: System performance and asynchronous communications I/O throughput can be significantly degraded by setting Receive Trigger Level to 1 on COM ports, whose serial controller device is capable of supporting Extended Hardware Buffering. This is particularly true where a COM port is sending and receiving data at high bit rates (over 2400 bps), or where multiple COM ports are simultaneously engaged in asynchronous communications.

Setting the Receive Trigger Level to 14 characters can increase the probability of getting receive hardware overrun errors. This is most likely where a COM port is sending and receiving data at high bit rates (over 2400 bps), or where multiple COM ports are simultaneously engaged in asynchronous communications.

“Function 73H – Query DCB Parameters” on page 18-50 always gives the actual current setting of the Receive Trigger Level, regardless of the setting of the Extended Hardware Buffering mode. In Enhanced mode, the Receive Trigger Level setting is ignored, and the ASYNC device driver automatically sets the value for maximum operational efficiency. In this case, the Receive Trigger Level chosen by the physical ASYNC device driver is not reflected in Function 73H. What the user has specified for the Receive Trigger Level in this IOCTL is returned, even though it is ignored.

Note 12: Transmit Buffer Load Count refers to the number of characters that the physical ASYNC device driver gives to a COM port's serial controller transmit hardware on the occurrence of a transmit hardware interrupt. On COM ports with a serial controller device that does not support Extended Hardware Buffering, bit 7 of DCB Flags3 is always set to 0, indicating that the device is in Character mode (Transmit Buffer Load Count = 1).

Normally, when the serial controller supports Extended Hardware Buffering and the physical device driver protocols are set to fully use this hardware capability, the Transmit Buffer Load Count is set to 16 characters. This means that every time the physical device driver gets control of the serial controller on the occurrence of a transmit hardware interrupt, 16 characters are placed in the serial controller's transmit buffer.

There might be situations where an application or communications subsystem needs to control the flow of data manually during a communications session. Setting the Transmit Buffer Load count to 1 character ensures that the physical device driver has control of the serial controller every time a character is transmitted.

When the physical device driver is operating in Automatic Protocol Override mode, setting the Transmit Buffer Load Count has no effect. This parameter setting is completely ignored unless DCB Flags3 bits 3 and 4 are set to Extended Hardware Buffering enabled.

Note: System performance and asynchronous communications I/O throughput can be significantly degraded by setting Transmit Buffer Load Count to 1 on COM ports, whose serial controller device is capable of supporting Extended Hardware Buffering. This is particularly true where a COM port is sending and receiving data at high bit rates (over 2400 bps), or where multiple COM ports are simultaneously engaged in asynchronous communications.

The IOCTL "Function 73H – Query DCB Parameters" on page 18-50 always gives the actual current setting of the Transmit Buffer Load Count, regardless of the setting of the Extended Hardware Buffering mode. In Enhanced mode, the Transmit Buffer Load Count setting is ignored, and the ASYNC device driver automatically sets the value for maximum operational efficiency. In this case, the Transmit Buffer Load Count chosen by the ASYNC device driver is not reflected in Function 73H. What the user has specified for the Transmit Buffer Load Count in this IOCTL is returned, even though it is ignored.

Function 54H – Set Enhanced Mode Parameters

This function sets the enhanced mode parameters.

Parameter Packet Format

Field	Length
Enhanced Flags1	BYTE
Reserved	DWORD

Enhanced Flags1 Has the following settings (see Note 1 for more information):

Bit 0 Enhanced mode supported by hardware. (Query only for Function 74H.)

Bit 1 Enable the enhanced mode (default)

Bits 2-3 DMA Receive Operation request. Has the following:

Bit 3	Bit 2	Description
0	0	Disable DMA Receive Capability
0	1	Enable DMA Receive Capability (default)
1	0	Dedicate a DMA channel to Receive operation
1	1	Reserved.

Bits 4-5 DMA Transmit operation request. Has the following:

Bit 5	Bit 4	Description
0	0	Disable DMA Transmit Capability
0	1	Enable DMA Transmit Capability (default)
1	0	Dedicate a DMA channel to Transmit operation
1	1	Reserved.

Bit 6 Receive operation in DMA mode. (Query only for Function 74H.)

Bit 7 Transmit operation in DMA mode. (Query only for Function 74H.)

Data Packet Format: None. Packet pointer must be NULL.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and none of the enhanced parameter settings for this COM device is changed.

Remarks: This function is used to disable/enable the Enhanced mode operations, or to control the Receive/Transmit operations on a COM port with the Enhanced UART, or compatibles. As a default, the physical device driver automatically runs I/O operations in Enhanced mode, which is either DMA mode or Enhanced FIFO mode. In DMA mode, the data transfer from port to memory, or from memory to port, can be done directly by the DMA chip, alleviating the CPU operations, and gaining maximum performance gain. A DMA-mode operation requires a system resource called the *DMA channel*.

In Enhanced FIFO mode, the full capacity of Extended Hardware Buffering is used automatically by the physical device driver. In both operations, the BIOS function interfaces are used by the physical device driver to maintain full compatibility with existing communication protocols. The Extended Hardware Buffering Flags3 bit settings of "Function 53H – Set DCB Parameters" are ignored.

The user can turn the Enhanced mode *off* with this function, if desired. In this case, a COM port behaves as if it is a conventional serial device with Extended Hardware Buffering capability. The Extended Hardware Buffering bit settings of Flags3 (Function 53H) are used to control the hardware FIFO buffer.

If a general failure error is not returned, then the actions described below are taken by the physical device driver. If the Data Packet pointer is not NULL, the IOctI fails with the ERROR_GEN_FAILURE return code.

To maintain hardware compatibility, the application must call “Function 74H – Query Enhanced Mode Parameters” on page 18-53, before the Function 54H is used. This allows the Reserved bits to be set correctly in a future release of the device driver. By calling Function 74H first, the application can maintain the state of the physical device driver for a mode that the application is not aware of.

Note 1: Enhanced Flags1. Bits 0, 6, and 7 are only for querying the status of the Enhanced mode (using Function 74H), and are ignored when the user calls Function 54H. Before setting bits 1-5, the user must ensure that a COM port supports the Enhanced mode, by calling Function 74H, and checking to see if bit 0 of the Enhanced Flags1 is set. If bit 1 is set in Function 54H, the physical device driver generates the general failure error return code.

As a default, when the Enhanced mode is supported, it is enabled, and the Enable Enhanced Mode bit is set. The user can disable the Enhanced mode by resetting this bit. When Enhanced mode is disabled, the settings of bits 2–5 are ignored, and the enhanced FIFO/DMA capabilities of the hardware are not exploited. It is recommended that the user not disable the Enhanced mode unless it is required to control the hardware FIFO buffer manually. In Enhanced mode, the physical device driver automatically controls the hardware FIFO buffer for its maximum efficiency. The advanced function interfaces provided by the hardware allow a full compatibility with the existing RS232-C communication protocols.

When the Disable DMA Receive Capability option is chosen, the device driver does not try to use the DMA capability for receive operations. Instead, the physical device driver runs receive operations in Enhanced FIFO mode where the full FIFO capability and the enhanced ABIOS function interfaces are automatically exploited. The efficiency of the Enhanced FIFO-mode operations is not as good as the DMA-mode operation, but is better than the Character-mode operation. This option allows the user to control the use of a limited number of the available DMA channels in the system. When the Enhanced mode is disabled, the system runs in conventional mode, and the advanced function features provided by the hardware are not utilized.

The initial default system uses the Enable DMA Receive Capability option. In Enhanced mode, the physical device driver tries to use a DMA channel at Open request time. If there is no DMA channel available at that moment, the operation defaults to Enhanced FIFO-mode operation. If the physical device driver can successfully allocate a DMA channel, then the receive operation operates in DMA mode. When the Enhanced mode is disabled, no attempt is made to allocate a DMA channel.

When the Dedicate a DMA Channel to Receive Operation option is requested, the physical device driver tries to allocate a DMA channel as a dedicated one for receive operations. If a DMA channel cannot be allocated at the time of request, the physical device driver returns the general failure error. If a DMA channel can be allocated successfully by Function 54H, it is not deallocated until the user issues Function 54H again, with another option such as Enable DMA Receive Operation, Disable DMA Receive Operation, or Disable Enhanced Mode.

Note: If the user requests to switch the state of DMA Receive Capability, while a receive operation is in process, there is a chance of loss of data. It is recommended that the user check the emptiness of the device driver receive and transmit queues through “Function 68H – Query Number of Characters in Receive Queue,” and “Function 69H – Query Number of Characters in Transmit Queue.” Before requesting the DMA-mode switch, the user must stop transmitting, and communication protocols must be employed to ensure the transmitting system on the other end of the line has stopped transmitting.

When the Disable DMA Transmit Capability option is chosen, the physical device driver does not try to use the DMA capability for transmit operations. Instead, the physical device driver runs transmit operation s in Enhanced FIFO-mode, where the full FIFO capability and the enhanced ABIOS function interfaces are automatically exploited. The efficiency of the Enhanced FIFO-mode operations is not as good as the

DMA-mode operation, but is better than the Character-mode operation. This option allows the user to control the use of a limited number of the available DMA channels in the system. When the Enhanced mode is disabled, the system runs in conventional mode, and the advanced function features provided by the hardware are not utilized.

The initial default system uses the Enable DMA Transmit Capability option. In Enhanced mode, with this option, the physical device driver tries to use a DMA channel at each transmit request, and if there is no DMA channel available at that moment, then the operation defaults to Enhanced FIFO-mode operation. If the physical device driver can successfully allocate a DMA channel, then the transmit operation operates in DMA mode. When the Enhanced mode is disabled, no attempt is made to allocate a DMA channel.

When the Dedicate a DMA Channel to Transmit Operation option is requested, the physical device driver tries to allocate a DMA channel as a dedicated one for transmit operation. If a DMA channel cannot be allocated at the time of request, the physical device driver returns the general failure error. If a DMA channel can be allocated successfully by Function 54H, it is not deallocated until the user issues Function 54H again, with another option such as Enable DMA Transmit Operation, Disable DMA Transmit Operation, or Disable Enhanced Mode.

Note: If the user requests to switch the state of DMA Transmit Capability, while a transmit operation is in process, there is a chance of loss of data. It is recommended that the user check the emptiness of the device driver transmit queue through “Function 69H – Query Number of Characters in Transmit Queue” on page 18-47

The First Level Open, or Last Level Close does not affect the bit settings of Enhanced Flags1.

Function 61H – Query Bit Rate

This function returns the bit rate.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Bit Rate	WORD

Bit Rate A binary integer representing the actual bit rate of the COM device in bits-per-second, rounded to the nearest whole number.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver returns the current bit rate of the COM device.

Note: If this function is called when the current bit rate setting of a COM port is greater than what can be stored in a 1-WORD field, the device driver sets the bit rate to 1200 bps (default value), and returns 1200 bps to the user.

Function 62H – Query Line Characteristics

This function returns the line characteristics (stop bits, parity, data bits, break).

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Data Bits	BYTE
Parity	BYTE
Stop Bits	BYTE
Transmitting Break	BYTE

Data Bits See “Function 42H – Set Line Characteristics” on page 18-9

Parity See Function 42H

Stop Bits See Function 42H

Transmitting Break 0 = not currently transmitting break. 1 = currently transmitting break.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure is not returned, the physical device driver returns the line characteristics as defined.

Function 63H – Extended Query Bit Rate

This function returns the extended query bit rate in doublewords for a bit rate higher than 19200 bps.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Current Bit Rate	DWORD
Fraction of Current Bit Rate	BYTE
Minimum Bit Rate Supported	DWORD
Fraction of Minimum Bit Rate Supported	BYTE
Maximum Bit Rate Supported	DWORD
Fraction of Maximum Bit Rate Supported.	BYTE

Current Bit Rate	The binary integer representing the actual bit rate in bits-per-seconds set for a COM port.
Fraction	The binary integer representing the fraction of the actual current bit rate set for a COM port.
Minimum Bit Rate	The binary integer representing the supportable minimum bit rate of a COM port in bits-per-second.
Fraction of Minimum	The binary integer representing the fraction of the supportable minimum bit rate for a COM port.
Maximum Bit Rate	The binary integer representing the supportable maximum bit rate in bits-per-seconds for a COM port. Depending on overall system overhead, and the electrical characteristics of the hardware cables and serial device adapter type, the actual value of the maximum bit rate supported might be lower than this.
Fraction of Maximum	The binary integer representing the fraction of the supportable maximum bit rate for the COM device.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure is not returned, the physical device driver returns the the bit rate values as defined. This function is extended from the current "Function 61H – Query Bit Rate" on page 18-39.

Function 64H – Query COM Status

This function returns the COM status.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
COM Status Byte	BYTE

COM Status Byte If equal to 1, the condition is TRUE. If equal to 0, the condition is FALSE.

Bit 0 Transmit status (Tx) waiting for CTS to be turned *on*. See Note 3 of “Function 53H – Set DCB Parameters” on page 18-21.

Bit 1 Tx waiting for DSR to be turned *on*. See Note 3 of Function 53H.

Bit 2 Tx waiting for DCD to be turned *on*. See Note 3 of Function 53H.

Bit 3 Tx waiting because XOFF received. Behaves as if XOFF received (Function 47H). See Note 2 of Function 53H.

Characters are transmitted immediately (Function 44H). When the device driver is in this state, it automatically transmits XONs and XOFFs due to Automatic Receive Flow Control.

Bit 4 Tx waiting because XOFF transmitted. See Note 2 of Function 53H.

Characters are still transmitted immediately. When the physical device driver is in this state, it automatically transmits XONs due to Automatic Receive Flow Control.

Bit 5 Tx waiting because break being transmitted. See “Function 4BH – Set Break ON” on page 18-20.

Bit 6 Character waiting to transmit immediately. See “Function 44H – Transmit Byte Immediate” on page 18-13.

Bit 7 Receive waiting for DSR to be turned *on*. See Note 4 of Function 53H.

Transmit status (Tx) indicates why transmission might not occur, regardless of whether there is data to transmit. However the device driver must be enabled for the given condition, for the status to reflect that the physical device driver is waiting for the given condition to transmit.

For example, 00000001 indicates that the physical device driver has put receive characters in the physical device driver receive queue, and is not waiting to transmit a character immediately (Function 44H). Characters are not transmitted from the physical device driver transmit queue when using CTS for output handshaking, because CTS does not have the proper value.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver returns the COM device current status.

Function 65H – Query Transmit Data Status

This function returns the transmit data status.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Transmit Status	BYTE

Transmit Status Returned as bit significant values. If the bit is 1, the condition is TRUE. If the bit is 0, the condition is FALSE. The number at the beginning of the description is the bit position number. The bit positions go from least to most significant.

- Bit 0** WRITE request packets in progress or queued
- Bit 1** Data in the physical device driver transmit queue
- Bit 2** Transmit hardware is currently transmitting data
- Bit 3** Character waiting to be transmitted immediately
- Bit 4** Waiting to automatically transmit an XON
- Bit 5** Waiting to automatically transmit an XOFF
- Bit 6** Undefined
- Bit 7** Undefined.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver returns the current transmit status of the COM device.

Function 66H – Query Modem Output Signals

This function returns the modem control output signals.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Modem Control Output Signals	BYTE

Modem Control Output If a bit value of 1, the condition is *on*. If a bit value of 0, the condition is *off*.

Bit 0 Data Terminal Ready (DTR)

Bit 1 Request To Send (RTS)

Bits 2-7 Undefined.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver returns the current modem control output signals of the COM device.

Function 67H – Query Modem Input Signals

This function returns the current modem control input signals.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Modem Control Input Signals	BYTE

Modem Control Input If a bit value of 1, the condition is *on*. If a bit value of 0, the condition is *off*.

Bits 0-3	Undefined
Bit 4	Clear To Send (CTS)
Bit 5	Data Set Ready (DSR)
Bit 6	Ring Indicator (RI)
Bit 7	Data Carrier Detect (DCD).

Returns: If the call is made with an invalid Parameter Packet value a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver returns the current modem control input signals of the COM device.

Function 68H – Query Number of Characters in Receive Queue

This function returns the number of characters in the receive queue.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Number of Characters Queued	WORD
Size of Receive Queue	WORD

Number of Characters Binary integer with the number of received characters in the device driver receive queue, which is a memory buffer between the memory pointed to by the READ request packet and the receive hardware for this COM device. The application cannot assume that there are no unsatisfied Read requests if there are characters in the device driver receive queue.

Note: The behavior of data movement between the Read request, and the receive queue can change with each release of the physical device driver. Applications should not have a dependency on this information.

Size of Receive Queue Binary integer with the size of the physical device driver receive queue. Applications should be independent of the receive queue being size (fixed or not fixed). The information in this field allows the application to get the size of the receive queue. The current size of the receive queue is approximately 1KB in PIO mode, and 2KB in DMA mode, but is subject to change. In DMA mode, the actual available space might be less than 2KB, since the operating system needs some marginal space (known as the *receive buffer threshold*) for the DMA operation.

Using this information, the application should avoid device driver receive queue overruns by using an application-to-application block protocol with the system communicating with the application.

Returns: If the call is made with an invalid Parameter Packet value, a general failure error is reported and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver returns the information.

Function 69H – Query Number of Characters in Transmit Queue

This function returns the number of characters in the transmit queue.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Number of Characters Queued	WORD
Size of Transmit Queue	WORD

Number of Characters Binary integer with the number of characters ready to be transmitted in the physical device driver transmit queue, which is a memory buffer between the memory pointed to by the WRITE request packet, and the transmit hardware for this COM device. If the transmit queue is empty, the application cannot assume that all Write requests have completed, or that no Write requests are outstanding.

Note: The behavior of data movement between the Write request, and the transmit queue can change with each release of the physical device driver. Applications should not be dependent on this information.

Size of Transmit Queue Binary integer with the size of the physical device driver transmit queue. Applications should be independent of the transmit queue size (fixed or not fixed). The information in this field allows the application to get the size of the transmit queue, which is 128 bytes in PIO mode, and 255 bytes in DMA mode, but is subject to change.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver returns the information.

Function 6DH – Query COM Error

This function returns COM Error (retrieves and clears the COM device error information).

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
COM Error WORD (COMERR)	WORD

COM Error WORD The appropriate bits in COM Error WORD are set by the physical device driver when the events described below occur. COM Error WORD is not cleared unless this function is performed by the physical device driver, or an OPEN request packet is received by the physical device driver and the COM device is not already open (First Level Open). See Note 5 of "Function 53H – Set DCB Parameters" on page 18-21.

- Bit 0** Receive queue overrun. No room in the physical device driver receive queue to put a character read in from the receive hardware.
- Bit 1** Receive hardware overrun. A character was not read from the hardware before the next character arrived, causing a character to be lost.
- Bit 2** The hardware detected a parity error.
- Bit 3** The hardware detected a framing error.
- Bits 4-15** Undefined.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, valid information is not returned in the Data Packet, and the COM Error WORD is not cleared.

Remarks: If a general failure error is not returned, the physical device driver returns and clears the COM device error information.

Function 72H – Query COM Event Information

This function returns the COM event information (retrieves and clears the COM device event WORD).

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
COM Event WORD	WORD

COM Event WORD The appropriate bits in the COM Event WORD are set by the device driver when the events described below occur. The COM Event WORD is not cleared unless this function is performed by the physical device driver, or an OPEN request packet is received by the physical device driver and the COM device is not already open.

- Bit 0** Set when any character is read from the COM device receive hardware and placed in the receive queue.
- Bit 1** Set whenever the serial port controller generates a Receive Timeout Interrupt during a Receive request. This bit is always zero, when the serial port controller does not support Extended Hardware Buffering.
- Bit 2** Set when the last character in the physical device driver transmit queue is sent to the COM device transmit hardware. Data in any outstanding Write requests still might need to be sent.
- Bit 3** Set when the Clear To Send (CTS) signal changes state.
- Bit 4** Set when the Data Set Ready (DSR) signal changes state.
- Bit 5** Set when the Data Carrier Detect (DCD) signal changes state.
- Bit 6** Set when a break is detected.
- Bit 7** Set when a parity, framing, or a receive hardware (or receive queue) overrun error occurs.
- Bit 8** Set when trailing edge of Ring Indicator is detected.
- Bits 9-15** Undefined.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, valid information is not returned in the Data Packet, and the event WORD is not cleared.

Remarks: If a general failure is not returned, the physical device driver returns the current value of the event WORD and then clears it.

Function 73H – Query DCB Parameters

This function returns Device Control Block (DCB) information.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Write Timeout	WORD
Read Timeout	WORD
Flags1	BYTE
Flags2	BYTE
Flags3	BYTE
Error Replacement Character	BYTE
Break Replacement Character	BYTE
XON Character	BYTE
XOFF Character	BYTE

- Write Timeout** Specifies the time period used for Write Timeout processing. See Note 8 of “Function 53H – Set DCB Parameters” on page 18-21. The value is in .01 second units based on zero (where 0 = .01 seconds).
- Read Timeout** Specifies the time period used for Read Timeout processing. See Note 9 of Function 53H. The value is in .01 second units based on zero (where 0 = .01 seconds).
- Flags1** Has the following bits:
- Bits 0-1** DTR Control mode. Has the following:
- | Bit 1 | Bit 0 | Description |
|-------|-------|--|
| 0 | 0 | Disable |
| 0 | 1 | Enable |
| 1 | 0 | Input handshaking |
| 1 | 1 | Invalid input. Results in a general failure error. |
- Bit 2** Reserved (returned as 0)
- Bit 3** Enable output handshaking using CTS
- Bit 4** Enable output handshaking using DSR
- Bit 5** Enable output handshaking using DCD
- Bit 6** Enable input sensitivity using DSR
- Bit 7** Reserved (returned as 0).
- Flags2** Has the following bits:
- Bit 0** Enable Automatic Transmit Flow Control (XON/XOFF)
- Bit 1** Enable Automatic Receive Flow Control (XON/XOFF)
- Bit 2** Enable error replacement character
- Bit 3** Enable null stripping (remove null bytes)

Bit 4 Enable break replacement character

Bit 5 Automatic Receive Flow Control.

0 = Normal

1 = Full-Duplex.

Bits 6-7 RTS Control mode. Has the following:

Bit 7	Bit 6	Description
0	0	Disable
0	1	Enable
1	0	Input handshaking
1	1	Toggling on transmit.

Flags3

Has the following bits:

Bit 0 Enable Write Infinite Timeout processing

Bits 1-2 Read Timeout processing. Has the following:

Bit 2	Bit 1	Description
0	1	Normal Read Timeout processing
1	0	Wait-For-Something, Read Timeout processing
1	1	No-Wait, Read Timeout processing.

Bits 3-4 Extended Hardware Buffering. Has the following:

Bit 4	Bit 3	Description
0	0	Not supported
0	1	Extended Hardware Buffering DISABLED
1	0	Extended Hardware Buffering ENABLED
1	1	Automatic Protocol Override.

Bits 5-6 Receive Trigger Level. Has the following:

Bit 6	Bit 5	Description
0	0	1 character
0	1	4 characters
1	0	8 characters
1	1	14 characters.

Bit 7 Transmit Buffer Load Count

0 = 1 character

1 = 16 characters.

See "Function 53H – Set DCB Parameters" on page 18-21 for field definitions.

Error Replacement Returned value. See Note 5 of Function 53H.

Break Replacement Returned value. See Note 7 of Function 53H.

XON Character Returned value. See Note 2 of Function 53H.

XOFF Character Returned value. See Note 2 of Function 53H.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, the physical device driver returns valid information in the Data Packet. The general Device Control Block (DCB) parameter access Functions 53H and 73H are used for:

- Automatic Transmit Flow Control (start/stop transmit when XON/XOFF character received)
- Automatic Receive Flow Control (transmit XON/XOFF when receive buffer fills or empties)
- Determining XON/XOFF characters
- DTR Control mode (enable/disable/input handshaking)
- RTS Control mode (enable/disable/input handshaking/toggling on transmit).
- Output handshaking using CTS/DSR/DCD (control signal determines when to transmit)
- Input sensitivity using DSR (reception of data controlled by DSR)
- Error Replacement character and processing
- Break Replacement character and processing
- Null stripping
- Receive/Transmit Timeout processing
- Extended Hardware Buffering control.

To maintain upward compatibility, it is the responsibility of the application to call “Function 73H – Query DCB Parameters” before calling “Function 53H – Set DCB Parameters.” The appropriate information in the returned control block should be modified by the application; then the Function 53H can be performed.

Note: The bit fields that are labeled *reserved* are returned as 0, but applications should not make this assumption or assume that the fourth bit combination will never be returned for DTR Control mode, Read Timeout processing, or Extended Hardware Buffering. Applications should not attempt to manipulate these reserved bits.

Function 74H – Query Enhanced Mode Parameters

This function returns the Enhanced mode parameters for a COM port. Also returns the state of the current DMA-operation mode, and user-requested settings for DMA receive/transmit operations.

Parameter Packet Format: None. Packet pointer must be NULL.

Data Packet Format

Field	Length
Enhanced Flags1	BYTE
Reserved	DWORD

Enhanced Flags1 Has the following settings (see Note 1 for more information):

Bit 0 Enhanced mode supported by hardware.

Bit 1 Enable the Enhanced mode (default)

Bits 2-3 DMA Receive Operation request. Has the following:

Bit 3	Bit 2	Description
0	0	Disable DMA Receive Capability
0	1	Enable DMA Receive Capability (Default)
1	0	Dedicate a DMA channel to Receive operation
1	1	Reserved.

Bits 4-5 DMA Transmit operation request. Has the following:

Bit 5	Bit 4	Description
0	0	Disable DMA Transmit Capability
0	1	Enable DMA Transmit Capability (Default)
1	0	Dedicate a DMA channel to Transmit operation
1	1	Reserved.

Bit 6 Receive operation in DMA mode.

Bit 7 Transmit operation in DMA mode.

Returns: If the call is made with an invalid Parameter Packet value, a *general failure* error is reported, and valid information is not returned in the Data Packet.

Remarks: If a general failure error is not returned, then the actions described below are taken by the physical device driver.

Note 1: Enhanced Flags1. Bit 1 represents the status of the Enhanced mode. As a default, the Enhanced mode is enabled and this bit is set to 1. Bits 2-5 represent the DMA operation options set by the user that have been executed successfully, or the system default, if no user options have been requested so far. The setting of bit 1 is only meaningful when bit 0 is set, and the settings of bits 2-5 are only meaningful when bit 1 is set.

Bit 6 shows the most recent Receive Operation mode. If it is set, the receive operation is in DMA mode. When the system is initialized and no receive operation has been done, the state of this bit is meaningless. Bit 7 shows the most recent Transmit Operation mode. If it is set, the transmit operation is in DMA mode. When the system is initialized and no transmit operation has been done, the state of this bit is meaningless.

Category 3 IOCtl Commands

The following is a summary of the Category 3 IOCtl Commands:

Function	Description
70H	Allocate an LDT Selector
71H	Deallocate an LDT Selector
72H	Query Pointer Draw Address.
73H	Initialize Call Vector Table
74H	ABIOS Pass-Through
75H	Allocate an LDT Selector with Offset
76H	Allocate an LDT Selector with Background Validation Options

Category 3 Pointer Draw Control IOCTL Command

The following is a description of the Category 3 Pointer Draw Control IOCTL Command:

Function	Description
72H	Query Pointer Draw Address.

Function 72H – Query Pointer Draw Address

This function returns the pointer draw address.

Parameter Packet Format: None.

Data Packet Format

Field	Length
Return Code	WORD
Pointer Draw Routine Entry Point (selector:offset)	DWORD
Pointer Draw Routine Data Segment Selector	WORD

Returns: The information in the Data Packet above.

Remarks: This function is used by the mouse subsystem to obtain the entry point address of the pointer draw routine. The pointer draw routine is contained within the physical Pointer Draw device driver. It is called by the physical Mouse device driver to update the pointer image on the screen. The FAR-16 address returned by Function 72H is passed by the mouse subsystem to the physical Mouse device driver through an IOCTL interface. See “Category 7 Mouse Control IOCTL Commands” on page 18-118. The physical Mouse device driver saves the FAR-16 address passed, and uses it when calling the pointer draw routine.

This function is supported by the physical Pointer Draw device driver.

Category 3 Video Control IOCTL Command

The following is a description of the Category 3 Video Control IOCTL Command:

Function	Description
73H	Initialize Call Vector Table.

Function 73H – Initialize Call Vector Table

This function initializes the Call Vector Table.

Parameter Packet Format: None.

Data Packet Format

Field	Length
Far Address of the Call Vector Table	DWORD

Returns: None.

Remarks: See the description of the Call Vector Table under “Video Device Handler Identification” on page 14-1.

Category 3 Screen Control IOCTL Commands

The following is a summary of the Category 3 Screen Control IOCTL Commands:

Function	Description
70H	Allocate an LDT Selector
71H	Deallocate an LDT Selector
74H	ABIOS Pass-Through
75H	Allocate an LDT Selector with Offset
76H	Allocate an LDT Selector with Background Validation Options

Function 70H – Allocate a Selector

This function allocates an LDT selector.

Parameter Packet Format

Field	Length
32-Bit Physical Address	DWORD
Length	WORD

Data Packet Format

Field	Length
LDT Selector	WORD

Returns: None.

Remarks: This function is supported by the Screen device driver. A 32-bit physical address and 16-bit length are passed to the physical device driver. The physical device driver returns an LDT selector for that area of memory.

Read/Write access is granted to any data areas completely contained in the range of addresses, from A0000 to BFFFF. Read-Only access is granted for all other data areas completely contained in the range of addresses, from 00000 to FFFFF. Requests for any other data areas results in a return code of ERROR_I24_INVALID_PARAMETER.

If the PhysToUVirt Device helper service returns a non-zero offset along with the requested selector, the selector is deallocated and ERROR_I24_INVALID_PARAMETER is returned. In this case, "Function 75H – Allocate a Selector with Offset" on page 18-63 should be used.

Function 71H – Deallocate a Selector

This function deallocates an LDT selector.

Parameter Packet Format

Field	Length
LDT Selector	WORD

Data Packet Format: None.

Returns: None.

Remarks: This function is supported by the Screen device driver. An LDT selector previously created by the Screen device driver by way of “Function 70H – Allocate a Selector” on page 18-60, is passed in the parameter block. The selector is deallocated.

Function 74H – BIOS Pass-Through

This function passes an BIOS request block to Unit 0 of the video device opened by SCREEN\$, and returns it to the caller on completion of the request.

Parameter Packet Format

Field	Length
BIOS Request Block	OTHER

Data Packet Format: None.

Returns: None.

Remarks: This function is supported by the physical Screen device driver. The parameter packet is copied to a 64-byte work area that is used as an BIOS request block. The LID and UNIT fields are overwritten with the Logical ID of the BIOS video device opened by SCREEN\$, and 0, respectively. This request block is used with the DevHlp, BIOSCall, and the resulting request block is copied back to the caller's Parameter Packet.

Function 75H – Allocate a Selector with Offset

This function allocates an LDT selector with offset.

Parameter Packet Format

Field	Length
32-Bit Physical Address	DWORD
Length	WORD

Data Packet Format

Field	Length
Offset within New LDT Selector	WORD
New LDT Selector	WORD

Returns: None.

Remarks: This function is supported by the physical Screen device driver. A 32-bit physical address and 16-bit length are passed to the device driver. The physical device driver returns an LDT selector and offset for that area of memory.

Read/Write access is granted to any data areas completely contained in the range of addresses, from A0000 to BFFFF. Read-Only access is granted for all other data areas completely contained in the range of addresses, from 00000 to FFFFF. Requests for any other data areas results in a return code of ERROR_I24_INVALID_PARAMETER.

Function 76H – Allocate a Selector with Background Validation

This function allocates an LDT selector with background validation.

Parameter Packet Format

Field	Length
32-Bit Physical Address	DWORD
Segment Length	WORD
Options	WORD

Options Characteristics of Video RAM (VRAM) address space are customized with the Options parameter. The following values are valid:

- 0** Make segment readable code; no validation in background
- 1** Make segment writable data; no validation in background
- 2** Free selector
- 3** Make segment readable IOPL code
- 4** Make segment Read/Write IOPL data
- 5** Tag selectors Physical Video Buffer (PVB); make segment writable data with validation in background
- Other** Reserved.

Data Packet Format

Field	Length
LDT Selector:Offset	DWORD

Returns: None.

Remarks: This function is supported by the physical Screen device driver. A 32-bit physical address and 16-bit length are passed to the physical device driver. The physical device driver returns an LDT selector and offset for that area of memory.

Read/Write access is granted to any data areas completely contained in the range of addresses, from A0000 to BFFFF. Read-Only access is granted for all other data areas completely contained in the range of addresses, from 00000 to FFFFF. Requests for any other data areas will return an error.

Category 4 Keyboard Control IOCTL Commands

The following is a summary of the Category 4 IOCTL Commands:

Function	Description
50H	Set Code Page
51H	Set Input Mode (Default ASCII)
52H	Set Interim Character Flags
53H	Set Shift State
54H	Set Typematic Rate and Delay
55H	Reserved
56H	Set Session Manager Hot Key
57H	Set KCB
58H	Set Code Page Number
59H	Set Read/Pek Notification
5AH	Alter Keyboard LEDs
5BH	Reserved
5CH	Set NLS and Custom Code Page
5DH	Create a New Logical Keyboard
5EH	Destroy a Logical Keyboard
71H	Query Input Mode
72H	Query Interim Character Flags
73H	Query Shift State
74H	Read Character Data Records
75H	Peek Character Data Record
76H	Query Session Manager Hot Key
77H	Query Keyboard Type
78H	Query Code Page Number
79H	Translate Scan Code to ASCII
7AH	Query Keyboard Hardware ID
7BH	Query Keyboard Code Page Support Information

Function 50H – Set Code Page

This function sets the code page.

Parameter Packet Format

Field	Length
Code Page Translation Table	OTHER

Code Page Translation Has the following format when there are 127 copies of the KeyDef record shown below (includes 1 for each possible scan code that can be returned from the keyboard). Not all entries are used; unused entries are zero. The entries are in scan code order, based on the remapped scan codes returned by the keyboard controller.

```

XlateTable:
  XHeader          : XHeader
  KeyDef1          : KeyDef
  KeyDef2          : KeyDef
  KeyDef3          : KeyDef
  .               :
  .               :
  .               :
  KeyDef127        : KeyDef
  AccentTbl        : AccentTable
End XlateTable

XHeader:
  XTableID         : WORD [Code Page Number]
  XTableFlags1     : Rec [Word Width]
                  : The following three bits determine which shift key
                  : or key combination affects CHAR3 of each KeyDef.
  ShiftAlt         : Bit 0 [Use Shift-Alt instead of Ctrl-Alt]
  AltGrafL         : Bit 1 [Use left Alt key as Alt-Graphics]
  AltGrafR         : Bit 2 [Use right Alt key as Alt-Graphics]
  ShiftLock        : Bit 3 [Treat Caps Lock as ShiftLock]
  DefaultTable     : Bit 4 [Default table for the Lang.]
  ShiftToggle      : Bit 5 [Toggle or Latch ShiftLock]
                  : When 1 toggle, else latch
  AccentPass       : Bit 6 [Pass accent and non-accent key through]
                  : When 1 pass on accent keys and beep, else beep only.
                  : The following four bits determine which shift key or key
                  : key combination causes Char5 to be used in each KeyDef.
  CapsShift        : Bit 7 [Caps-Shift uses CHAR5]
  MachDep          : Bit 8 [Machine-dependent table]
  Reserved         : Bits 9-10
  Reserved         : Bits 11-15
EndRec XtableFlags1

```

```

XTableFlags2      : Rec[WORD Width]
Reserved          : Bits 0-15
EndRec XtableFlags2

KbdType           : WORD [Keyboard type, see below]
KbdSubType        : WORD [Reserved]
XtableLen         : WORD [Length of table]
EntryCount        : WORD [Number of KeyDef entries]
EntryWidth        : WORD [Width of KeyDef entries]
Country           : WORD [Language ID]
TableTypeID       : WORD [Identifies the table type]
                   1st byte (type)   : 01X 00X
                   2nd byte (sub-type): 00X Reserved
SubCountryID      : 4 Bytes [Sub-language Identifier]
Reserved          : 8 WORDS [Reserved]
End XHeader

KeyDef = Rec      [127 copies of this record.]
XlateOp = Rec     [WORD field] [Translate operation specifier.]
AccentFlags       : 7 Bits [See Notes 1 and 8.]
KeyType           : 9 bits [See Note 2.]
Char1             : Byte [Use depends on KeyType, see below.]
Char2             : Byte [Use depends on KeyType, see below.]
Char3             : Byte [Use depends on KeyType, see below.]
Char4             : Byte [Use depends on KeyType, see below.]
Char5             : Byte [Use depends on KeyType, see below.]
EndRec KeyDef

AccentTable = Rec [Table of accent key definitions.]
AccentEntry1      : AccentEntry
AccentEntry2      : AccentEntry
.
.
.
AccentEntry7      : AccentEntry
EndRec AccentTable

AccentEntry = Rec [Accent entry definition. See Notes 1 and 9.]
NonAccent         : 2 Bytes [Char/scan code when not used as accent]
CtlAccent         : 2 Bytes [Char/scan code when used with CTL.]
AltAccent         : 2 Bytes [Char/scan code when used with Alt.]
Map1              : 2 Bytes [From char-to-char for translation.]
Map2              : 2 Bytes " " " "
.
.
.
Map20             : 2 Bytes " " " "
EndRec AccentEntry

TableTypeID
1st Byte         2nd Byte
type             sub-type
00X             Reserved
0S/2            01X      00X

```

Data Packet Format: None.

Returns: None.

Remarks: This request changes the device driver resident code page for the system, and updates the zero entry of the Code Page Control Block.

Note1: The AccentFlags field of the KeyDef record has seven flags that are individually set, if a corresponding entry in the accent table applies to this scan code. If the key pressed immediately before the current one was an accent key, and the bit for that accent is set in the AccentFlags field for the current key, the corresponding AccentTable entry is searched for the replacement character value to use. If no replacement is found, and bit 6 of the XlateFlags1 field is set, the *not-an-accent* beep is sounded and the accent character, and current character, are passed as two separate characters. Also see Note 8.

Note 2: The KeyType field of the KeyDef record currently has the following values defined. The remaining values up to 1FH are undefined. The effect of each type of shift is defined below. Except where otherwise noted, when no shifts are active, Char1 is the translated character. (See Note 3.) Notice that any of the Alt, Alt + Char, Alt + Shift, or Alt + Gr keys (or all of them) can be present on a keyboard based on the AltGrafL and AltGrafR bits in the XTableFlags1 flag WORD in the table header.

01H AlphaKey. Alphabetical character key:

Shift	Uses Char2. If Caps Lock, uses Char1.
Caps Lock	Uses Char2. If Shift, uses Char1.
Ctrl	Set standard control code for this key's Char1 value. See Note 4.
Alt	Standard extended code. See Note 7.
Alt + Char	Uses Char3, if it is not 0.
Alt + Shift	Uses Char3, if it is not 0.
Alt + Gr	Uses Char3, if it is not 0.

02H SpecKey. Special non-alphabetical character key, no Caps Lock or Alt:

Shift	Uses Char2
Caps Lock	No effect, only depends on Shift, or Ctrl
Ctrl	See Note 4.
Alt	Marked undefined
Alt + Char	Uses Char3, if it is not 0
Alt + Shift	Uses Char3, if it is not 0
Alt + Gr	Uses Char3, if it is not 0.

03H SpecKeyC. Special non-alpha character key with Caps Lock. See Note 15.

Shift	Uses Char2. If Caps Lock, uses Char1.
Caps Lock	Uses Char2. If Shift, uses Char1.
Ctrl	See Note 4.
Alt	Uses Char4, if not zero. See Note 7.
Alt + Char	Uses Char3, if it is not 0.
Alt + Shift	Uses Char3, if it is not 0.
Alt + Gr	Uses Char3, if it is not 0.

04H SpecKeyA. Special non-alpha character key, with Alt (no Caps Lock):

Shift	Uses Char2
Caps Lock	No effect; depends on Shift, Ctrl, or Alt only
Ctrl	See Notes 5 and 9
Alt	See Notes 7 and 10
Alt + Char	Uses Char3, if it is not 0
Alt + Shift	Uses Char3, if it is not 0
Alt + Gr	Uses Char3, if it is not 0.

05H SpecKeyCA. Special non-alpha character key, with Caps Lock and Alt:

Shift	Uses Char2. If Caps Lock, uses Char1.
Caps Lock	Uses Char2. If Shift, uses Char1.
Ctrl	See Note 4.
Alt	See Note 7.
Alt + Char	Uses Char3, if it is not 0.
Alt + Shift	Uses Char3, if it is not 0.

Alt + Gr Uses Char3, if it is not 0.

06H **FuncKey.** Function keys. Char1 = *n* in *F_n*; Char2 ignored. Sets extended codes 58 + Char1, if no shift; if F11 or F12, uses 139 and 140.

Shift Sets extended codes 83 + Char1. F11 and F12 use 141 and 142, respectively.

Caps Lock No effect on function keys.

Ctrl Sets extended codes 93 + Char1. F11 and F12 use 143 and 144, respectively.

Alt Sets extended codes 103 + Char1. F11 and F12 use 145 and 146, respectively.

Alt + Char Uses Char3, if it is not 0.

Alt + Shift Uses Char3, if it is not 0.

Alt + Gr Uses Char3, if it is not 0.

07H **PadKey.** Keypad keys (see Note 5 for definition of Char1). Note that non-shifted use of these keys is fixed to the extended codes:

Shift Uses Char2, unless Num Lock. See Note 5.

Caps Lock No effect on pad keys, unless Num Lock. See Note 5.

Ctrl Sets extended codes. See Note 5.

Alt Used to build a character. See Note 5.

Alt + Char Uses Char3, if it is not 0.

Alt + Shift Uses Char3, if it is not 0.

Alt + Gr Uses Char3, if it is not 0.

08H **SpecCtlKey.** Special action keys, when used with Ctrl down:

Shift No effect on these keys

Caps Lock No effect on these keys

Ctrl Uses Char2

Alt See Note 7

Alt + Char Uses Char3, if it is not 0

Alt + Shift Uses Char3, if it is not 0

Alt + Gr Uses Char3, if it is not 0.

09H **PrtSc.** Print Screen key; sets Char1 normally (see Note 17):

Shift Signal the Print Screen function

Caps Lock No effect on this key

Ctrl Sets extended code and signals the Print Echo function

Alt Marked undefined

Alt + Char Uses Char3, if it is not 0

Alt + Shift Uses Char3, if it is not 0

Alt + Gr Uses Char3, if it is not 0.

0AH **SysReq.** System Request key; treated like a shift key. See Note 6.

0BH **AccentKey.** Keys that affect the next key pressed (also known as *dead keys*). Char1 is an index into the AccentTbl field of the XlateTable, selecting the AccentEntry that corresponds to this key. Char2 and Char3 do the same for the shifted Accent character. See Note 15.

Shift Uses Char2 to index to applicable AccentEntry.

Caps Lock No effect on this key.

Ctrl Uses CtlAccent character from AccentEntry. See Note 8.

Alt Uses AltAccent character from AccentEntry. See Note 8.

Alt + Char Uses Char3 to index to applicable AccentEntry.

Alt + Shift Uses Char3 to index to applicable AccentEntry.

Alt + Gr Uses Char3 to index to applicable AccentEntry.

Note: Key types, 0CH - 13H, set Char1 and Char2 to mask values as defined in Note 6.

- 0CH** ShiftKeys. Shift or Ctrl key, sets and clears flags. Char1 holds the bits in the lower byte of the shift status WORD to set when the key is down, and clear when the key is released. Char2 does the same thing for the upper byte of the shift status WORD unless the secondary key prefix (hex E0) is seen immediately prior to this key, in which case Char3 is used in place of Char2.
- 0DH** ToggleKey. General toggle key (like Caps Lock). Char1 holds the bits in the lower byte of the shift status WORD to toggle on the first make of the key after it is pressed. Char2 holds the bits in the upper byte of the shift status WORD to set when the key is down, and clear when the key is released unless the secondary key prefix (hex E0) is seen immediately prior to this key, in which case Char3 is used in place of Char2.
- 0EH** AltKey. Treated just like ShiftKeys above, but has its own key type, because when seen, the accumulator used for Alt-PadKey entry is zeroed to prepare such entry (see Note 5). Sometimes this key is treated as the AltC/S/G key (that is, either Alt + Char, Alt + Shift, or Alt + Gr), if one of the AltGraf bits is *on* in XTableFlags1.
- 0FH** Num Lock. Normally behaves like ToggleKey, but the physical Keyboard device driver sets a pause screen indication when this key is used with the Ctrl key depressed. The pause is cleared on the following keystroke, if that stroke is a character generating key.
- 10H** Caps Lock. This key is treated as a type 0DH toggle key. It has a separate entry here so that it can be processed like a Shift Lock key when that flag is set in the XTableFlags1 WORD in the header. When treated as a Shift Lock, the Caps Lock flag in the shift status WORD is set *on* on any make of this key, and only cleared when the left or right shift key is depressed. Char2 and Char3 are processed the same as ToggleKey.
- 11H** Scroll Lock. Normally behaves like ToggleKey but has a separate entry here. When used with Ctrl —, it can be recognized as Ctrl-Break.
- 12H** XShiftKey. Extended Shift Key (for Country support). See Note 9.
- 13H** XToggleKey. Extended Toggle Key (for Country support). See Note 9.
- 14H** SpecKeyCS. Special key 1 for foreign keyboard processing. See Note 15.
- | | |
|---------------------|-------------|
| Shift | Uses Char2 |
| Caps Lock | Uses Char4 |
| Ctrl | See Note 4 |
| Alt | See Note 7 |
| Alt + Char | Uses Char3 |
| Alt + Shift | Uses Char3 |
| Alt + Gr | Uses Char3 |
| Caps + Shift | Uses Char5. |
- 15H** SpecKeyAS. Special key 2 for foreign keyboard processing. See Note 15.
- | | |
|--------------------|---------------------------|
| Shift | Uses Char2. |
| Caps Lock | No effect on this key. |
| Ctrl | See Note 4. |
| Alt | Uses Char 4. See Note 14. |
| Alt + Char | Uses Char 3. See Note 14. |
| Alt + Shift | Uses Char 3. See Note 14. |
| Alt + Gr | Uses Char 3. See Note 14. |
- 1AH** Extended Extended key. This corresponds to the BIOS level support provided for INT 16H, Functions 20H, 21H, and 22H.
- | | |
|--------------------|-----------------------|
| Shift | Uses Char2 |
| Caps Lock | No effect on this key |
| Ctrl | Uses Char4 |
| Alt | Uses Char5 |
| Alt + Char | Uses Char 3, if not 0 |
| Alt + Shift | Uses Char 3, if not 0 |

Alt + Gr Uses Char 3, if not 0.

16-1FFH Reserved, except for 1AH, the Extended Extended key (see above).

Note 3: Undefined Character Code. Any key combination that does not fall into any of the defined categories. For example, the Ctrl key pressed along with a key that has no defined control mapping, is mapped to the value 0, and the key type is set in the KeyPacket record indicating undefined translation. The KeyPacket record passed to the monitors, if installed, contain the original scan code in the ScanCode field, and the 0 in the Character field, for this key. Notice that no character data records with an undefined character code are placed in the keyboard input buffer.

Note 4: Ctrl Key. The six possible situations that can occur when a key is pressed with only the Ctrl shift key are shown below:

- The key pressed is an AlphaKey character. In this case, the Ctrl plus Char1 combination defines one of the standard defined control codes (all numbers are decimal):

Ctrl-	Mapping	Code Name	Ctrl-	Mapping	Code Name
a	1	SOH	n	14	SO
b	2	STX	o	15	SI
c	3	ETX	p	16	DLE
d	4	EOT	q	17	DC1
e	5	ENQ	r	18	DC2
f	6	ACK	s	19	DC3
g	7	BEL	t	20	DC4
h	8	BS	u	21	NAK
i	9	HT	v	22	SYN
j	10	LF	w	23	ETB
k	11	VT	x	24	CAN
l	12	FF	y	25	EM
m	13	CR	z	26	SUB

Notice that any key defined as AlphaKey uses the Char1 code value minus 96 (ASCII code for a) plus 1, to set the mapping shown above. Any scan code defined as AlphaKey, must assign to Char1 one of the allowed lower case letters.

- The key pressed is a non-alpha character, such as [, but is not an action key, such as Enter, Backspace, or an arrow key. This is a SpecKey[C][A] in the list of key types above. In this case, with one exception, the mapping is based on the scan code of the key. Though the key can be re-labeled, the Ctrl + Char combination is always mapped based on the scan code of the key using the following table (all numbers are decimal):

Scan Code	US Kbd Legend	Mapped Value	Name of New Code
3	2 @	0	Null
7	6 ^	30	RS
12	-	31	US (see Note below)
26	[{	27	Esc
27] }	29	GS
43	\	28	FS

Note: The mapping for the hyphen character (-) is the one exception. The scan code for it is ignored, only the ASCII code for hyphen (decimal 45) is looked for in Char1 when mapping the Ctrl + - combination. This is because there can be more than one occurrence of the hyphen (-) key on the keyboard. The Ctrl + - (PadKey minus) combination produces character/scan code values of 00/8EH, respectively.

- The key pressed is an action key like Enter, Backspace, or an arrow key. These keys generate special values when used in conjunction with the Ctrl key. Those actions are defined in other notes where they apply. Two particular keys in this category are:

Category 4 – Keyboard IOCTls

Ctrl+Enter = LF(010)
Ctrl+Backspace = Del(127).

- The key pressed is a function key, F1 - F12. See the FuncKey description in Note 2.
- The key pressed is an accent key. See Note 8.
- The key is not defined in conjunction with Ctrl. In this case, the key is treated as undefined, as described in Note 3.

Note 5: PadKey. The pad keys have several uses depending on various shift states. Some of them are based on their position on the keyboard. Because keyboard layouts change, the hard-coded assumed positions of the keypad keys, with the offset value that must be coded into Char1, are defined below. Any remapping must use the Char1 values shown below for the keys that correspond to the pad keys given by the Legend, or Char2 values:

US Kbd Legend	Scan Code	Char1 Required	Char2 US Kbd	With Ctrl
Home	7	71	Decimal 0	ASCII 7
Up	8	72	" 1	" 8
PgUp	9	73	" 2	" 9
-	74	" 3	" -	" 12
Left	4	75	" 4	" 4
5	76	" 5	" 5	" 5
Right	6	77	" 6	" 6
+	78	" 7	" +	" 7
End	1	79	" 8	" 1
Down	2	80	" 9	" 2
PgDn	3	81	" 10	" 3
Ins	0	82	" 11	" 0
Del	.	83	" 12	" .

Notice that when Num Lock is *off*, or if Shift is active and Num Lock *on*, the code returned is the extended code. The code returned corresponds to the Legends above (Home, PgUp, and so forth). When Num Lock is *on*, or if Shift is active and Num Lock is *off*, the code returned is Char2. Notice that the +, and - keys also return Char2, when the shift key is down.

When the Alt key is used with the PadKeys, the absolute value of the pressed key (looked up using the required Char1 value) is added to the accumulated value of any of the previous numeric keys pressed, without releasing the Alt key. Before adding the new number to the accumulated value that accumulation is multiplied by ten, with overflow beyond 255 ignored. When Alt is released, the accumulation becomes a Character code, and is passed along with a scan code of zero. Notice that if any key other than the 10 numeric keys is hit, the accumulated value is reset to zero. When the keypad *, -, or + keys are pressed while the Alt key is down, the extended characters 55, 74, and 78 (decimal) are returned, respectively.

When AltGraphics is used with the PadKeys, the Char3 value is returned if it is non-zero, and if an AltGraf bit is set in XTableFlags1; otherwise it is treated the same as the Alt key.

On the Enhanced keyboard, the secondary keypad keys return, as an extended character, the scan code of the key plus 80 (decimal) when pressed in conjunction with the Alt key. The secondary / key returns an extended character of 164, when pressed in conjunction with the Alt key.

Note 6: State Key. Each state key entry has Char1, Char2, and Char3 defined as follows:

- **Char1.** A mask to set the appropriate bit in the low byte of the keyboard Shift Flags when the state key is pressed. When the state key is a toggle key, the set bit is toggled each additional time the key is pressed. When the state key is not a toggle key, the set bit is cleared when the key is released.
- **Char2.** A mask to set the appropriate bit in the high byte of the Keyboard Shift Flags when the key is pressed.

- **Char3.** Used in place of Char2 when the secondary key prefix is seen immediately prior to this key.

The masks are shown below (numbers are in hex):

Key	Char1	Char2	Char3
-----	-----	-----	-----
Right Shift	01	00	00
Left Shift	02	00	00
Ctrl Shift	04	01	04
Alt Shift	08	02	08
Scroll Lock	10	10	10
Num Lock	20	20	20
Caps Lock	40	40	40
SysReq	00	80	80

Notice that the INS key is not treated as a state key, but as a pad key. Also, SysReq is included here since it is treated as a shift key.

Note 7: Alt Character. Most of the keys defined in a category that allows the Alt key (AlphaKey, SpecKeyA, SpecKeyCA) return a value called an *extended character*. This value is a character code of 00H, or E0H, with a second byte (using the ScanCode field of the CharData record) defining the extended code. In most cases, this value is the scan code of the key. Since the legend on these keys can be remapped on a foreign language keyboard, the Alt-based extended code is hard to define in a general sense. The following rules are used:

- **AlphaKey.** The extended code is derived from Char1 (the lower-case character) as it was originally mapped on the PC keyboard. The original scan code value is the extended code that a character returns. These keys can be moved and will still return their original Alt extended codes.
- **SpecKeyA and SpecKeyCA.** This category is used for all keys that are not an alphabetical character or an action code (like Enter or Backspace, the only exception being the Tab key, which is treated as a character). On foreign keyboards, these keys can be moved around and can have new values assigned to them, such as special punctuation symbols. Therefore, the Alt mappings must be based on the real scan code as any keys defined by the SpecKey_ classification will only have an Alt mapping, if it is in one of the positions defined below. In that case, the Alt extended code is as shown:

Scan Code	US Kbd Legend	Alt Value	Scan Code	US Kbd Legend	Alt Value
-----	-----	-----	-----	-----	-----
2	1 !	120	15	Tab	165
3	2 @	121	26	[{	26
4	3 #	122	27] }	27
5	4 \$	123	28	Enter	28
6	5 %	124	39	; :	39
7	6 ^	125	40	' "	40
8	7 &	126	41	` ~	41
9	8 *	127	43	\	43 (equals W.T.C key number 42)
10	9 (128	51	, <	51
11	0)	129	52	. >	52
12	- _	130	53	/ ?	53
13	= +	131			

The secondary / key returns an extended character of 164 when pressed while Alt is down.

- **FuncKey.** Defined in Note 2.
- **SpecCtlKey.** The Alt+ values of the Escape, Backspace, and Enter keys are extended characters equaling 1, 14, and 28 (decimal), respectively.

When AltGraphics is used the Char3 value is returned if it is non-zero, and if an AltGraf bit is set in XTableFlags1. Otherwise, it is treated the same as the Alt key.

Note 8: Accent Key. When an accent key is pressed with Ctrl, or Alt, it is treated as a regular key. The character it translates to is the one in the CtlAccent or AltAccent field of the AccentEntry pointed to by the Char5 value of the KeyDef. If the key being defined has no defined value with Ctrl or Alt, it should have zeroes in the field of the undefined combination.

When an accent key is pressed by itself (or with Right Shift, Left Shift, or AltGraphics), it is not translated immediately. The Char1 (or Char2, when Left or Right Shift or AltGraphics is used), index in the KeyDef record is used with the next key received to check if the next key has an accent mapping. If that next key has no mapping for this accent (that is, if it has no bit set in its AccentFlags), or if that next key is not found in this accent's AccentEntry, then the character value in the NonAccent field of the AccentEntry is used as the character to display followed by the translation of that next key after the not-an-accent beep is sounded.

Notice that if a key doesn't change when a Left or Right Shift key is pressed, it should use the same value for Char1 and Char2 so that the accent applies in both the shifted and non-shifted cases. If the accent value is undefined when used with a shift key, or AltGraphics, the value in Char2 or Char3 should be 0.

Any accent key that doesn't have an Alt or Ctrl mapping should put zeros in the AltAccent and CtlAccent fields of its AccentEntry. If the value in the table is between 1 and 7, then the key is considered an accent key and further accent key processing is indicated. See Note 1 for more information.

Note 9: Extended State Key. For special Country support, the Keyboard device driver maintains another byte of shift status. Key types 12H and 13H are provided for manipulation of that byte. The other fields of the KeyDef are:

- **Char1.** A mask in which bits that are *on*, are those bits that define the field being used for the Char2 value. Only bits in the NLS shift status byte that correspond to the bits in this byte are altered by the Char2 value.
- **Char2.** For KeyType 12H (Extended Shift), the value to OR into the byte when the make code is seen. Also, the inverted value is ANDed when the break code is seen. For KeyType 13H (Extended Toggle), the value XORed into the byte on each make code seen (break code ignored).
- **Char3.** Use in place of the Char2 when the secondary key prefix (hex E0) is seen immediately prior to this key.

For example, Char1 or Char2 can define single shift status bits to set/clear/toggle. Char2 can be a set of coded bits, delineated by Char1, that are set to a numeric value when the key is hit, and cleared to zero when released (or on the next hit, if toggled). The whole byte can be set to Char2 when Char1 has all bits *on*.

Note 10: Space Key. The key treated as the space character should have a flag set in its AccentFlags field for each possible accent, that is, for each defined AccentEntry in the AccentTable. And each AccentEntry should have the Space character defined as one of its accented characters, with the translation having the same value as the accent character itself. The reason for this is that, by definition, an Accent Key followed by the space character maps to the accent character alone. If the table is not set up as just described, a not-an-accent beep is sounded whenever the accent key followed by a space is pressed.

Notice that the space key is defined as a SpecKeyA (type 4) because its use, in conjunction with the Alt key, is allowed. In this case, and when used with the Ctrl key, it returns the ASCII space character. This works correctly, except in the case of the dieresis accent (double-dot) in code page 437. The space is treated as an invalid character and the beep result occurs, with the dieresis represented by double quote. Characters are displayed depending upon the language in effect when the invalid dieresis is encountered. For some languages, the character substituted is the double-quote; for others, the character used is the F9H character.

Note 11: KbdType identifies the hardware-specific keyboard used by this table. The values and allowable types are the same as specified in “Function 77H – Query Keyboard Type” on page 18-99.

Note 12: The DefaultTable flag in XtableFlags1 is used by the KEYB utility in loading code pages when changing from one language to another. It identifies the default code page to KEYB, should KEYB not find one or both CODEPAGE = defined code pages.

Note 13: The Language IDs and Sub-country IDs used are as follows:

Keyboard Layout Country Code	Keyboard Layout SubCountry Code	Country
AR	785	Arabic-speaking
BE	120	Belgium
CF	058	Canadian-French
CS	243	Czechoslovakia
CS	245	Czechoslovakia
DK	159	Denmark
SU	153	Finland
FR	120	France
FR	189	France
GR	129	Germany
HE	972	Hebrew-speaking
HU	208	Hungary
IS	197	Iceland
IT	141	Italy
IT	142	Italy
LA	171	Latin-American Spanish
NL	143	Netherlands
NO	155	Norway
PL	214	Poland
PO	163	Portugal
SP	172	Spain
SV	153	Sweden
SF	150F	Swiss-French
SG	150G	Swiss-German
TR	179	Turkey
UK	166	United Kingdom
UK	168	United Kingdom
US	103	United States
YU	234	Yugoslavia

Note 14: Keytype 15. When the Alt, or Alt+Shift, key is pressed, both XlatedChar and XlatedScan, in the CharData record will have the same value.

Category 4 – Keyboard IOctls

Note 15: If the Charx value is in the range of 1-7, then Charx identifies an accent key. Otherwise, Charx is treated as a valid ASCII character. This does not apply to Ctrl-Charx sequences.

Note 16: If Alt+Gr, Alt+Shift, or Alt+Ctrl is pressed, and Char3 is 0, the Alt key is used to translate to a valid result.

Note 17: The * key on the keypad of the Enhanced keyboard, although producing the same scan code/character as that of the regular IBM Personal Computer AT* keyboard, is treated differently since a dedicated Print Screen key exists on the Enhanced keyboard. The following scan codes/characters are returned by the physical Keyboard device driver for the Enhanced keyboard * key on the keypad:

Unshifted	37H/2AH
Shifted	37H/2AH
Ctrl	96H/00
Alt	37H/00

Note 18: Size. The code page described here has the following dimensions:

Xlate Header	=	40
127 KeyDefs @ 7 bytes	=	889
7 AccentEntries @ 46 bytes	=	322

		1251 bytes

Returns: None.

Remarks: This request changes the device driver resident code page for the system. This IOctl updates the 0 entry of the Code Page Control Block.

* Trademark of the IBM Corporation

Function 51H – Set Input Mode

This function sets the input mode.

Parameter Packet Format

Field	Length
Mode	BYTE

Mode A 1-byte field containing:

Bit 1xxxxx1 Shift Report
Bit 0xxxxx0 ASCII mode
Bit 1xxxxxx BINARY mode.

Data Packet Format: None.

Returns: The error return codes for this function are as follows:

8113H INVALID_PARAMETER
 810CH GENERAL_FAILURE.

Remarks: This request is used to pass the current input mode to the Keyboard device driver. The Keyboard device driver maintains the mode separately for each session. The caller can interrogate the mode using “Function 71H – Query Input Mode” on page 18-90. The mode is also returned on every call to “Function 74H – Read Character Data Records” on page 18-93, and “Function 75H – Peek Character Data Record” on page 18-95. The default input mode is ASCII. The physical device driver uses the mode when processing CTL functions and reporting the shift state, if Shift Report is set *on*.

Function 52H – Set Interim Character Flags

This function sets the interim character flags.

Parameter Packet Format

Field	Length
Flag	BYTE

Flag A 1-byte field containing bit flags. A bit set to 1, indicates the state listed below:

- Bit 7** Interim console flag *on*.
- Bit 6** Reserved. Must equal 0.
- Bit 5** Program requested on-the-spot conversion.
- Bit 4** Reserved. Must equal 0.
- Bit 3** Reserved. Must equal 0.
- Bit 2** Reserved. Must equal 0.
- Bit 1** Reserved. Must equal 0.
- Bit 0** Reserved. Must equal 0.

Data Packet Format: None.

Returns: The error return code for this function is as follows:

8113H INVALID_PARAMETER

Remarks: This request is used to set the interim character flags maintained by the physical Keyboard device driver. The physical Keyboard device driver maintains the flags separately for each session, and passes the interim character flags with each character data record to keyboard monitors.

Function 53H – Set Shift State

This function sets the shift state.

Parameter Packet Format

Field	Length
Shift State	WORD
NLS	BYTE

Shift State A WORD field containing shift states:

High Byte Has the following settings:

- Bit 15** SysReq key down
- Bit 14** Caps Lock key down
- Bit 13** NumLock key down
- Bit 12** ScrollLock key down
- Bit 11** Right Alt key down
- Bit 10** Right Ctrl key down
- Bit 9** Left Alt key down
- Bit 8** Left Ctrl key down.

Low Byte Has the following:

- Bit 7** Insert on
- Bit 6** Caps Lock on
- Bit 5** NumLock on
- Bit 4** ScrollLock on
- Bit 3** Either Alt key down
- Bit 2** Either Ctrl key down
- Bit 1** Left Shift key down
- Bit 0** Right Shift key down.

NLS A byte field containing NLS shift status.

Data Packet Format: None.

Returns: None.

Remarks: This request is used to set the current shift state for the keyboard. The physical Keyboard device driver maintains the shift state separately for each logical keyboard. Notice that this call overrides the Shift State set by previous shift keystrokes. Also, the Shift State set by this function code is overridden by any subsequent shift keystrokes. This shift state is inserted into the Character Data Record that is built for each incoming keystroke.

Function 54H – Set Typematic Rate and Delay

This function sets typematic rate and delay.

Parameter Packet Format

Field	Length
Delay	WORD
Rate	WORD

Delay Specifies the typematic delay in milliseconds. A value greater than the maximum value defaults to the maximum value.

Rate Specifies the typematic rate in characters per second. A value greater than the maximum value defaults to the maximum value.

Data Packet Format: None.

Returns: None.

Remarks: This request is used to set the keyboard typematic rate and delay to the values specified in the request.

Function 56H – Set Session Manager Hot Key

This function sets the Session Manager hot key.

Parameter Packet Format

Field	Length
State	WORD
Make Code	BYTE
Break Code	BYTE
Hot Key ID	WORD

State Has the following settings:

High Byte Bit settings are as follows:

- Bit 15** Reserved = 0
- Bit 14** Reserved = 0
- Bit 13** Reserved = 0
- Bit 12** Reserved = 0
- Bit 11** Right Alt key down
- Bit 10** Right Ctrl key down
- Bit 9** Left Alt key down
- Bit 8** Left Ctrl key down.

Low Byte Bit settings are as follows:

- Bit 7** Reserved = 0
- Bit 6** Reserved = 0
- Bit 5** Reserved = 0
- Bit 4** Reserved = 0
- Bit 3** Reserved = 0
- Bit 2** Reserved = 0
- Bit 1** Left Shift key down
- Bit 0** Right Shift key down.

Make Code The scan code of the hot key make

Break Code The scan code of the hot key break

Hot Key ID The Hot Key ID. Value is set by the caller. Notice that ID value FFFFH is reserved and must not be used as a Hot Key ID. See **Remarks**.

A maximum of two of the above bit selections can be selected for a given hot key definition. If more than two bits are selected, or if a reserved bit is selected, the result is an `INVALID_PARAMETER` error code returned to the caller.

Data Packet Format: None.

Returns: The error return codes for this function are as follows:

- 8103H `BAD_COMMAND`.
- 8113H `INVALID_PARAMETER`

Remarks: This request is used by the Session Manager to set a list of keyboard hot keys, searched for by the physical Keyboard device driver. Up to 16 hot keys can be defined by the Session Manager for handling by the physical Keyboard device driver. The new hot key is global, that is, it applies to all sessions. A hot key can be redefined by calling this function with the same Hot Key ID.

The combination of shift flags in the first WORD, and scan codes in the second, allow the Session Manager to set hot key combinations such as Alt + Esc. The hot key is triggered on detection of the scan code for the hot key break.

Note: If a DOS application has claimed hardware INT 9 or INT 50, the hot key is triggered on detection of the break scan code for the required shift key.

Hot key definition requests, which have identical data definitions (shift state, make, and break scan codes) but different Hot Key IDs, result in an `INVALID_PARAMETER` error code returned to the caller. Attempts to define a new or unique hot key when the maximum number is currently active, also result in an `INVALID_PARAMETER` error code returned to the caller.

This IOCTL is successful only if performed by the process which initially invoked it, unless it is called with a Hot Key ID of `-1`. A Hot Key ID with a value of `-1` acts like a toggle. The first time a call is made with `-1`, the Ctrl + Alt + Del, Ctrl + Esc, and Alt + Esc keystrokes are disabled. The second time a call is made with `-1`, the keystroke sequences are enabled.

The caller of this interface must use caution in the key combinations selected since hot key definitions become system property, and any resulting characters normally produced by that key combination will no longer be available to applications.

Function 57H – Set KCB

This function sets KCB.

Parameter Packet Format

Field	Length
KCB Handle	WORD

KCB Handle The handle identifying the logical keyboard's KCB. A value of 0 in this parameter indicates the default keyboard.

Data Packet Format: None.

Returns: The error return codes for this function are as follows:

8103H BAD_COMMAND

810CH GENERAL_FAILURE.

Remarks: This request binds the specified logical keyboard (KCB) to the physical keyboard for this session. This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

Function 58H – Set Code Page Number

This function sets the code page number.

Parameter Packet Format

Field	Length
Code Page Number	WORD
Code Page Layer	WORD

Code Page Number 1-WORD containing the current code page number.

Code Page Layer Has the following settings:

- 0 Primary Code Page Layer
- 1 Secondary Code Page Layer.

Data Packet Format: None.

Returns: The error return codes for this function are as follows:

810CH GENERAL_FAILURE.
8113H INVALID_PARAMETER

Remarks: Sets the code page translation table, used by the current KCB, to the code page translation table identified by the input parameter. This IOCTL is can be called from a DOS Session.

Function 59H – Set Read/Peek Notification

This function sets read/peek notification.

Parameter Packet Format

Field	Length
Session Number	WORD
Read/Peek Notification Flags	WORD

Session Number Indicates the session to be enabled or disabled.

Read/Peek Flags Contains the following flags:

Bits 15-1 Reserved. Set to 0.

Bit 0 Mask indicating whether to enable or disable Read/Peek notification:

0 Disable Read/Peek notification

1 Enable Read/Peek notification.

Data Packet Format: None.

Returns: The error return codes for this function are as follows:

8103H BAD_COMMAND.

8113H INVALID_PARAMETER

Remarks: Sets the DDFlags of a key packet, instructing the Keyboard device driver to send a special monitor packet down the monitor chain. This packet is sent on every Read request seen by the physical Keyboard device driver for a particular screen group, and continues until the screen group is terminated or a subsequent call disables Read Notification.

The Process ID (PID) of the first caller of this IOCTL in the system is saved by the physical Keyboard device driver. Subsequent invocations of this IOCTL by a different PID results in an UNKNOWN_COMMAND error.

Function 5AH – Alter Keyboard LEDs

This function alters the keyboard LEDs.

Parameter Packet Format

Field	Length
State of Each LED	WORD

State of Each LED A bit mask indicating the new LED indicators state. A set bit means a particular LED is turned *on*. A clear bit indicates the LED is turned *off*. The bit mask is defined as follows:

Bits 15-3 Reserved. Set to 0.

Bit 2 If set, Caps Lock LED turned *on*. If clear, Caps Lock LED turned *off*.

Bit 1 If set, Num Lock LED turned *on*. If clear, Num Lock LED turned *off*.

Bit 0 If set, Scroll Lock LED turned *on*. If clear, Scroll Lock LED turned *off*.

If any of the reserved bits are set, an `INVALID_PARAMETER` error is returned.

Data Packet Format: None.

Returns: The error return codes for this function are as follows:

8103H `BAD_COMMAND`.

8113H `INVALID_PARAMETER`

Remarks: Alters the keyboard LEDs without changing the operational state of the keyboard's mode of scan code generation. This IOCTL is reserved for use by the Presentation Manager session system, and cannot be called by applications. Requests from applications return an `UNKNOWN_COMMAND` error.

Because it is not possible to separate the keyboard's LED and operational state for the 88/89 Key Enhanced Keyboard, the physical Keyboard device driver performs no operations for IOCTL requests received when the 88/89 Key Enhanced keyboard is the attached system keyboard.

Function 5CH – Set NLS and Custom Code Page

This function sets the National Language Support (NLS) and custom code page.

Parameter Packet Format

Field	Length
Code Page Translation Table Pointer	DWORD
Code Page Number	WORD
Index to Load.	WORD

- Code Page Table Pointer** The selector:offset pointing to the Code Page Translation Table. Notice that the physical Keyboard device driver does not perform machine model, or submodel determination, to validate this translate table address for a given machine or keyboard. This determination is the responsibility of the keyboard subsystem and system initialization routines.
- Code Page Number** 1-WORD identifying the code page number.
- Index to Load** 1-WORD identifying the number of the Code Page Control Block to load (1 or 2). A -1 indicates a custom code page for which the segment containing the custom code page is locked. This option is not valid for the DOS Session.

Data Packet Format: None.

Returns: The error return codes for this function are as follows:

810CH GENERAL_FAILURE.
8113H INVALID_PARAMETER

Remarks: This request is used to install one of two possible Code Page Translation Tables into the device driver. This IOCTL updates the number 1 (or 2) entry of the Code Page Control Block. Entry zero is the device driver resident Code Page Translation Table.

Notice that this IOCTL is similar to "Function 50H – Set Code Page" on page 18-66, except that different entries in the Code Page Control Block are updated. This IOCTL can be called from a DOS Session.

Function 5DH – Create a Logical Keyboard

This function creates a new logical keyboard.

Parameter Packet Format

Field	Length
KCB ID	WORD
Code Page ID	WORD

KCB ID A WORD containing a handle (from DosOpenHandle) used to identify the new logical keyboard.

Code Page ID A WORD containing the desired initial code page to be used for the new logical keyboard.

Data Packet Format: None.

Returns: The error return codes for this function are as follows:

8103H BAD_COMMAND
810CH GENERAL_FAILURE.
8113H INVALID_PARAMETER

Remarks: The desired initial code page should be either the default code page for the system, or one of the two optional code pages that can be prepared in the CODEPAGE= statement in CONFIG.SYS.

This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

Function 5EH – Destroy a Logical Keyboard

This function destroys an existing logical keyboard.

Parameter Packet Format

Field	Length
KCB ID	WORD

KCB ID A WORD containing a unique value used to identify the logical keyboard. A zero indicates the default keyboard.

Data Packet Format: None.

Returns: The error return code for this function is as follows:

810CH GENERAL_FAILURE.

Remarks: This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

Function 71H – Query Input Mode

This function returns the input mode.

Parameter Packet Format: None.

Data Packet Format

Field	Length
Mode	BYTE

Mode A 1-byte field containing one of the following values:

Bit 1xxxxxx1 Shift report
Bit 0xxxxxx0 ASCII mode
Bit 1xxxxxxx BINARY mode.

Returns: None.

Remarks: This request is used to obtain the input mode of the session of the currently active process. The input mode can be set with “Function 51H – Set Input Mode” on page 18-77. The input mode is meaningful for Ctrl-C, Ctrl-P, Ctrl-S, Ctrl-Break, Ctrl-ScrollLock, and Ctrl-PrtSc processing only.

Function 72H – Query Interim Character Flags

This function returns the interim character flags.

Parameter Packet Format: None.

Data Packet Format

Field	Length
Flags	BYTE

Flags A 1-byte field containing flag bits. A bit set to 1, indicate the state listed below:

- Bit 7** Interim console flag *on*
- Bit 6** Reserved = 0
- Bit 5** Program requested on-the-spot conversion
- Bit 4** Reserved = 0
- Bit 3** Reserved = 0
- Bit 2** Reserved = 0
- Bit 1** Reserved = 0
- Bit 0** Reserved = 0.

Returns: None.

Remarks: This request is used to obtain the interim character flags maintained by the physical Keyboard device driver.

Function 73H – Query Shift State

This function returns the shift state.

Parameter Packet Format: None.

Data Packet Format

Field	Length
Shift State	WORD
NLS	BYTE

Shift State A WORD field containing shift states:

High Byte Has the following settings:

- Bit 15** SysReq key down
- Bit 14** Caps Lock key down
- Bit 13** NumLock key down
- Bit 12** ScrollLock key down
- Bit 11** Right Alt key down
- Bit 10** Right Ctrl key down
- Bit 9** Left Alt key down
- Bit 8** Left Ctrl key down.

Low Byte Has the following:

- Bit 7** Insert *on*
- Bit 6** Caps Lock *on*
- Bit 5** NumLock *on*
- Bit 4** ScrollLock *on*
- Bit 3** Either Alt key down
- Bit 2** Either Ctrl key down
- Bit 1** Left Shift key down
- Bit 0** Right Shift key down.

NLS A byte field containing NLS shift status.

Returns: None.

Remarks: This request is used to obtain the shift state of the session of the currently active process. The shift state is set by incoming key strokes, and by calling “Function 53H – Set Shift State” on page 18-79.

Function 74H – Read Character Data Records

This function reads Character Data Records.

Parameter Packet Format

Field	Length
Transfer Count	WORD

Transfer Count A 1-WORD field containing the record transfer count. The sign bit of this WORD is set to request one of the following actions:

- 0 Wait for the requested number of key strokes to become available. The physical device driver blocks the requestor until all requested Character Data Records are available, and have been transferred to the caller.
- 1 Do not wait for the requested number of key strokes to become available. In this case, all characters currently available are transferred up to the requested transfer count.

Data Packet Format

Field	Length
CharData Record	10 BYTES

CharData Record The character data structure of the API function, KbdCharIn, is shown below:

- ASCIICode (UCHAR)** ASCII character code. The scan code received from the keyboard is translated to the ASCII character code.
- ScanCode (UCHAR)** Code received from the keyboard. Scan code received from the keyboard is translated to the ASCII character code.
- Status (UCHAR)** State of the keystroke event:
- Bits 7-6** Has the following values:
 - 00 Undefined
 - 01 Final character; interim character flag is turned off
 - 10 Interim character
 - 11 Final character; interim character flag is turned on.
 - Bit 5** If set to 1, immediate conversion requested.
 - Bits 4-2** Reserved.
 - Bit 1** Has the following values:
 - 0 Scan code is a character
 - 1 Scan code is not a character; instead it is an extended key code from the keyboard.
 - Bit 0** If set to 1, shift status returned without a character.
- Reserved (UCHAR)** NLS shift status. Reserved; must be set to 0.
- ShiftKeyStat (USHORT)** Shift key status. Values are:
- Bit 15** SysReq key down
 - Bit 14** Caps Lock key down

Bit 13	NumLock key down
Bit 12	Scroll Lock key down
Bit 11	Right Alt key down
Bit 10	Right Ctrl key down
Bit 9	Left Alt key down
Bit 8	Left Ctrl key down
Bit 7	Insert <i>on</i>
Bit 6	Caps Lock <i>on</i>
Bit 5	NumLock <i>on</i> .
Bit 4	Scroll Lock <i>on</i>
Bit 3	Either Alt key down
Bit 2	Either Ctrl key down
Bit 1	Left Shift key down
Bit 0	Right Shift key down

Time

Time stamp indicating when a key was pressed. It is specified in milliseconds from the time the system was started.

Returns: The error return codes for this function are as follows:

8103H BAD_COMMAND
8111H CHAR_CALL_INTERRUPTED
8113H INVALID_PARAMETER.

Remarks: This request is used to obtain one or more character data records from the Keyboard Input buffer (KIB) for the session of the currently active process. Notice that if shift report is *on*, then the CharData Record might not contain a character; instead, it could contain a shift state change in the Shift Status field.

This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

Function 75H – Peek Character Data Record

This function peeks at a Character Data Record.

Parameter Packet Format

Field	Length
Status	WORD

Status A 1-WORD field which contains 0 to indicate no key stroke is available, or 1 to indicate that a Character data Record is being returned. The sign bit is set to 0, if the current input mode is ASCII, or 1, if the input mode is binary.

Data Packet Format

Field	Length
CharData Record	10 BYTES

CharData Record The character data structure of the API function, KbdCharIn, is shown below:

ASCIICharCode (UCHAR)	ASCII character code. The scan code received from the keyboard is translated to the ASCII character code.																						
ScanCode (UCHAR)	Code received from the keyboard. Scan code received from the keyboard is translated to the ASCII character code.																						
Status (UCHAR)	State of the keystroke event: <table> <tr> <td>Bits 7-6</td><td>Has the following values: <table> <tr> <td>00</td><td>Undefined</td></tr> <tr> <td>01</td><td>Final character; interim character flag is turned off</td></tr> <tr> <td>10</td><td>Interim character</td></tr> <tr> <td>11</td><td>Final character; interim character flag is turned on.</td></tr> </table> </td></tr> <tr> <td>Bit 5</td><td>If set to 1, immediate conversion requested.</td></tr> <tr> <td>Bits 4-2</td><td>Reserved.</td></tr> <tr> <td>Bit 1</td><td>Has the following values: <table> <tr> <td>0</td><td>Scan code is a character</td></tr> <tr> <td>1</td><td>Scan code is not a character; instead it is an extended key code from the keyboard.</td></tr> </table> </td></tr> <tr> <td>Bit 0</td><td>If set to 1, shift status returned without a character.</td></tr> </table>	Bits 7-6	Has the following values: <table> <tr> <td>00</td><td>Undefined</td></tr> <tr> <td>01</td><td>Final character; interim character flag is turned off</td></tr> <tr> <td>10</td><td>Interim character</td></tr> <tr> <td>11</td><td>Final character; interim character flag is turned on.</td></tr> </table>	00	Undefined	01	Final character; interim character flag is turned off	10	Interim character	11	Final character; interim character flag is turned on.	Bit 5	If set to 1, immediate conversion requested.	Bits 4-2	Reserved.	Bit 1	Has the following values: <table> <tr> <td>0</td><td>Scan code is a character</td></tr> <tr> <td>1</td><td>Scan code is not a character; instead it is an extended key code from the keyboard.</td></tr> </table>	0	Scan code is a character	1	Scan code is not a character; instead it is an extended key code from the keyboard.	Bit 0	If set to 1, shift status returned without a character.
Bits 7-6	Has the following values: <table> <tr> <td>00</td><td>Undefined</td></tr> <tr> <td>01</td><td>Final character; interim character flag is turned off</td></tr> <tr> <td>10</td><td>Interim character</td></tr> <tr> <td>11</td><td>Final character; interim character flag is turned on.</td></tr> </table>	00	Undefined	01	Final character; interim character flag is turned off	10	Interim character	11	Final character; interim character flag is turned on.														
00	Undefined																						
01	Final character; interim character flag is turned off																						
10	Interim character																						
11	Final character; interim character flag is turned on.																						
Bit 5	If set to 1, immediate conversion requested.																						
Bits 4-2	Reserved.																						
Bit 1	Has the following values: <table> <tr> <td>0</td><td>Scan code is a character</td></tr> <tr> <td>1</td><td>Scan code is not a character; instead it is an extended key code from the keyboard.</td></tr> </table>	0	Scan code is a character	1	Scan code is not a character; instead it is an extended key code from the keyboard.																		
0	Scan code is a character																						
1	Scan code is not a character; instead it is an extended key code from the keyboard.																						
Bit 0	If set to 1, shift status returned without a character.																						
Reserved (UCHAR)	NLS shift status. Reserved; must be set to 0.																						
ShiftKeyStat (USHORT)	Shift key status. Values are: <table> <tr> <td>Bit 15</td><td>SysReq key down</td></tr> <tr> <td>Bit 14</td><td>Caps Lock key down</td></tr> <tr> <td>Bit 13</td><td>NumLock key down</td></tr> <tr> <td>Bit 12</td><td>Scroll Lock key down</td></tr> <tr> <td>Bit 11</td><td>Right Alt key down</td></tr> <tr> <td>Bit 10</td><td>Right Ctrl key down</td></tr> <tr> <td>Bit 9</td><td>Left Alt key down</td></tr> <tr> <td>Bit 8</td><td>Left Ctrl key down</td></tr> </table>	Bit 15	SysReq key down	Bit 14	Caps Lock key down	Bit 13	NumLock key down	Bit 12	Scroll Lock key down	Bit 11	Right Alt key down	Bit 10	Right Ctrl key down	Bit 9	Left Alt key down	Bit 8	Left Ctrl key down						
Bit 15	SysReq key down																						
Bit 14	Caps Lock key down																						
Bit 13	NumLock key down																						
Bit 12	Scroll Lock key down																						
Bit 11	Right Alt key down																						
Bit 10	Right Ctrl key down																						
Bit 9	Left Alt key down																						
Bit 8	Left Ctrl key down																						

Bit 7	Insert <i>on</i>
Bit 6	Caps Lock <i>on</i>
Bit 5	NumLock <i>on</i> .
Bit 4	Scroll Lock <i>on</i>
Bit 3	Either Alt key down
Bit 2	Either Ctrl key down
Bit 1	Left Shift key down
Bit 0	Right Shift key down

Time

Time stamp indicating when a key was pressed. It is specified in milliseconds from the time the system was started.

Returns: The error return codes for this function are as follows:

8103H	BAD_COMMAND
8111H	CHAR_CALL_INTERRUPTED
8113H	INVALID_PARAMETER.

Remarks: This request is used to obtain one CharData Record from the head of the Keyboard Input Buffer (KIB) of the session for the currently active process. The CharData Record is not removed from the KIB. Notice that if shift report is *on*, then the CharData Record might not contain a character; instead it could contain a shift state change in the Shift Status field.

This function returns an UNKNOWN_COMMAND error if the caller is in the Presentation Manager session.

Function 76H – Query Session Manager Hot Key

This function returns the Session Manager hot key.

Parameter Packet Format

Field	Length
Type	WORD

Type A 1-WORD subtype indicating the type of information to return:

- 0** Return the maximum number of hot keys the physical Keyboard device driver can support. This number is returned in the Parameter Packet.
- 1** Return the number of hot keys currently defined for the system (this number is returned in the Parameter Packet), and return the Session Manager hot key information for each number (this information is returned in the Data Packet buffer).

If the parameter list on entry is *1*, one or more hot key data structures are returned in the Data Packet format.

Data Packet Format

Field	Length
State	WORD
Make Code	BYTE
Break Code	BYTE
Hot Key ID	WORD

State Has the following settings:

High Byte Bit settings are as follows:

- Bit 15** Reserved = 0
- Bit 14** Reserved = 0
- Bit 13** Reserved = 0
- Bit 12** Reserved = 0
- Bit 11** Right Alt key down
- Bit 10** Right Ctrl key down
- Bit 9** Left Alt key down
- Bit 8** Left Ctrl key down.

Low Byte Bit settings are as follows:

- Bit 7** Reserved = 0
- Bit 6** Reserved = 0
- Bit 5** Reserved = 0
- Bit 4** Reserved = 0
- Bit 3** Reserved = 0
- Bit 2** Reserved = 0
- Bit 1** Left Shift key down
- Bit 0** Right Shift key down.

Make Code The scan code of the hot key make

Category 4 – Keyboard IOCTLs

Break Code The scan code of the hot key break

Hot Key ID The Hot Key ID. Value is set by the caller. Notice that ID value FFFFH is reserved and should not be used.

A maximum of two of the above bit selections can be chosen for a given hot key definition. If more than two bits are selected, or if a reserved bit is selected, the result is an INVALID_PARAMETER error code returned to the caller.

Returns: The error return code for this function is as follows:

8113H INVALID_PARAMETER.

Remarks: This request is used to obtain the scan code the physical Keyboard device driver uses as the Session Manager hot key, and should first be called with parameter list subtype=0 to determine the maximum number of hot keys the physical device driver can support. The value returned is used to determine the required size of the data buffer on a subsequent call to return the hot key data structures (parameter list subtype=1).

Function 77H – Query Keyboard Type

This function returns a keyboard type.

Parameter Packet Format: None.

Data Packet Format

Field	Length
Keyboard Type	WORD
Reserved = 0	DWORD

Keyboard Type A 1-WORD field containing:

High Byte Reserved = 0.

Low Byte Has the following bit values:

00H	Personal Computer AT keyboard*
01H	Enhanced Keyboard
02H-FFH	Reserved = 0.

Returns: None.

Remarks: This request returns the keyboard type. The 101 and 102 Key Enhanced keyboards, 88 and 89 Key Enhanced keyboards, 122 Key Enhanced keyboards, and MFI keyboards all return the value 01H to indicate an Enhanced keyboard is attached. For specific keyboard Hardware IDs, see “Function 7AH – Query Keyboard Hardware ID” on page 18-103.

* Trademark of the IBM Corporation

Function 78H – Query Code Page Number

This function returns the code page number.

Parameter Packet Format

Field	Length
Code Page Number	WORD
Reserved. Set to 0.	WORD

Code Page Number A WORD containing the current code page number in use. Values other than Code Page IDs that can be returned are as follows:

- 0 Indicates that PC US 437 is used
- 1 Indicates that a custom code page is installed.

Data Packet Format: None.

Returns: None.

Remarks: This request returns the code page in use by the current KCB. This IOCTL can be called from a DOS Session.

Function 79H – Translate Scan Code to ASCII

This function translates scan code to ASCII.

Parameter Packet Format

Field	Length
Code Page Number	WORD
Reserved. Set to 0.	WORD

Code Page Number 1-WORD containing the current code page number.

Data Packet Format

Field	Length
CharData Record	10 BYTES
KbdDD Flags	WORD
Xlate Flags	WORD
Xlate State1	WORD
Xlate State2	WORD

CharData Record The character data structure of the API function, KbdCharIn, is shown below:

ASCIICharCode (UCHAR) ASCII character code. The scan code received from the keyboard is translated to the ASCII character code.

ScanCode (UCHAR) Code received from the keyboard. Scan code received from the keyboard is translated to the ASCII character code.

Status (UCHAR) State of the keystroke event:

Bits 7-6 Has the following values:

00 Undefined

01 Final character; interim character flag is turned *off*

10 Interim character

11 Final character; interim character flag is turned *on*.

Bit 5 If set to 1, immediate conversion requested.

Bits 4-2 Reserved.

Bit 1 Has the following values:

0 Scan code is a character

1 Scan code is not a character; instead it is an extended key code from the keyboard.

Bit 0 If set to 1, shift status returned without a character.

Reserved (UCHAR) NLS shift status. Reserved; must be set to 0.

ShiftKeyStat (USHORT) Shift key status. Values are:

Bit 15 SysReq key down

Bit 14 Caps Lock key down

Bit 13 NumLock key down

Bit 12	Scroll Lock key down
Bit 11	Right Alt key down
Bit 10	Right Ctrl key down
Bit 9	Left Alt key down
Bit 8	Left Ctrl key down
Bit 7	Insert on
Bit 6	Caps Lock on
Bit 5	NumLock on.
Bit 4	Scroll Lock on
Bit 3	Either Alt key down
Bit 2	Either Ctrl key down
Bit 1	Left Shift key down
Bit 0	Right Shift key down

Time Time stamp indicating when a key was pressed. It is specified in milliseconds from the time the system was started.

KbdDD Flags Defined in the Device Monitor packets. See Chapter 11, “Physical Keyboard Device Driver” on page 11-1.

Xlate Flags Has the following:

High Byte Bit settings are as follows:

Bits 8-15 Reserved = 0.

Low Byte Bit settings are as follows:

Bits 1-7 Reserved = 0

Bit 0 Translation complete.

Xlate State1 Identifies the state of translation across successive calls. Initially this WORD should be zero and should be reset to 0 when the caller wants a new start to translation. Notice that it can take several calls to this IOCTL to complete a character, so this field should not be touched unless a fresh start to translation is desired. Also, this field is set to 0 at the completion of translation.

Xlate State2 Same as Xlate State1.

Returns: The error return code for this function is as follows:

8113H INVALID_PARAMETER.

Remarks: This request translates a scan code in a CharData Record to an ASCII character. Optionally, a code page can be specified to use for translation. Otherwise, the code page of the active KCB is used. To translate a given scan code with a particular shift state, indicate the shift state desired with the Shift State field of the CharData Record.

Function 7AH – Query Keyboard Hardware ID

This function returns the Hardware ID for the currently attached keyboard.

Parameter Packet Format: None.

Data Packet Format

Field	Length
Data Length	WORD
Hardware ID	WORD

Data Length On input, Data Length specifies the maximum number of returned data bytes requested by the caller. The Length value on input includes the length field's size. An input length field value of less than 2 is returned to the caller with an `INVALID_PARAMETER` error.

On output, Data Length always indicates the maximum number of bytes that this function can return. For OS/2 2.0, this return length size is 4 bytes.

Hardware ID The attached Hardware ID.

Returns: Returns the Hardware ID for the currently attached keyboard. The error return code for this function is as follows:

8113H `INVALID_PARAMETER`.

Remarks: In the past, all keyboards could be supported by knowing the hardware family information available through "Function 77H – Query Keyboard Type" on page 18-99. The 122-key keyboard has a number of differences from the other 88/89 Key Enhanced Family keyboards. Therefore, applications performing keystroke-specific functions might need to determine which keyboard is attached.

OS/2 system-supported keyboards, and their hardware generated IDs, are as follows:

Keyboard	Hardware ID
PC AT* Standard Keyboard	0001H
101 Key Enhanced Keyboard	AB41H
102 Key Enhanced Keyboard	AB41H
88 Key Enhanced Keyboard	AB54H
89 Key Enhanced Keyboard	AB54H
122 Key Mainframe Interactive (MFI) Keyboard	AB86H

* Trademark of the IBM Corporation

Function 7BH – Query Keyboard Code Page Information

This function returns the keyboard current code page support information.

Parameter Packet Format: None.

Data Packet Format

Field	Length
Data Length	WORD
KbdCP	WORD
Ctry	ASCIIZ
SCtry	ASCIIZ

Data Length The input return data length requested, or the output data structure length required, in bytes. On input, the length field value specifies the caller's return data area size. A minimum value of 2 is required so that the length value can be properly returned. An `INVALID_PARM` error code is returned for input length values less than 2.

On output, the remaining returned data depends on the input length field value. Length field values of 12 bytes or greater result in the full data structure being returned. For length requests less than 12, only the portion of returned data fields (which can be completely returned including ASCIIZ string terminators) is returned. The Length field on return always represents the byte count of returned data including string terminators.

KbdCP The active keyboard code page number.

Ctry The active keyboard Country code in ASCIIZ string format.

SCtry The active keyboard Subcountry code in ASCIIZ string format.

Returns: The error return code for this function is as follows:

8113H `INVALID_PARAMETER`.

Remarks: This request returns the keyboard's current code page support information. This information is particularly useful for utilities like *KEYB*. The code page support information consists of the keyboard's active code page, and the Country and Subcountry codes active for the keyboard layout. These items are the system-level support factors used by the physical Keyboard device driver for global translation.

Category 5 Parallel Port Control IOCTL Commands

The following is a summary of Category 5 IOCTL Commands:

Function	Description
42H	Set Frame Control (CPL, LPI)
44H	Set Infinite Retry
45H	Reserved
46H	Initialize Parallel Port
47H	Reserved
48H	Activate Font
4BH	Reserved
4CH	Reserved
4DH	Set Print-Job Title
4EH	Set Parallel Port IRQ Time-Out Value
4FH	Reserved
62H	Query Frame Control
64H	Query Infinite Retry
66H	Query Parallel Port Status
67H	Reserved
69H	Query Active Font
6AH	Verify Font
6BH	Reserved
6CH	Reserved
6DH	Reserved
6EH	Query Parallel Port IRQ Time-Out Value
6FH	Reserved.

Function 42H – Set Frame Control

This function sets frame control.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Characters per Line	BYTE
Lines per Inch	BYTE

Characters per Line Valid numbers are 80 and 132.

Lines per Inch Valid numbers are 6 and 8.

Returns: The error return codes for this function are as follows:

- 0100H Completed successfully
- 8102H Device not ready, if monitors not registered
- 8103H Invalid command, if CPL/LPI is not (80,6), (80,8), (132,6), or (132,8)
- 810AH Write fault, if monitor registered and the DevHlp, MonWrite, fails

Function 44H – Set Infinite Retry

This function sets infinite retry.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Data	BYTE

Data The data is defined as:

- 0 Disable infinite retry
- 1 Enable infinite retry.

Returns: The error return codes for this function are as follows:

- 0100H Completed successfully.
- 8103H Invalid command, if data not 0 or 1.

Remarks: If a monitor is registered for the physical Parallel Port device driver, infinite retry is always enabled. When a monitor is registered, a disable request returns a good return code and the function is not performed.

Function 46H – Initialize Parallel Port

This function initializes a parallel port.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Set to zero	BYTE

Returns: The error return codes for this function are as follows:

0100H Completed successfully

810AH Write fault, if monitor registered and the DevHlp, MonWrite, fails.

Function 48H – Activate Font

This function activates a font.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Code Page	WORD
Font ID	WORD

Code Page The value of the code page to make the currently active code page.

0000H If the Code Page value and Font ID are specified as zero, set printer to hardware default code page and font.

0001H – FFFFH Valid code page numbers.

Font ID The ID value of the font to make currently active.

0000H If the Code Page value and Font ID are specified as zero, set printer to hardware default code page and font. If Font ID is zero and the code page is a valid non-zero, then any font within the specified code page is acceptable.

0001H – FFFFH Valid Font ID numbers; font types defined by the font file definitions for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label, and are also entered in the DEVINFO statement for the printer.

Returns: The error return codes for this function are as follows:

0100H	Completed successfully
810AH	Write fault if monitor registered and the DevHlp, MonWrite, fails
C102H	Code page is not available
C103H	No code page function because spooler not started
C104H	Font ID is not available (verify)
C109H	Error caused by switcher error not by input parameters
C10AH	Error caused by invalid printer name as input
C10DH	Received code page request when code page switcher not initialized
C10FH	SFN table full, cannot activate another entry
C113H	DASD error reading font file
C115H	DASD error reading font file definition block
C117H	DASD error while writing to temporary spool file
C118H	Disk full error while writing to temporary spool file
C119H	Spool file handle was bad.

Function 4DH – Set Print-Job Title

This function sets the print-job title.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Length	WORD
Address of ASCIIZ Job Title	DWORD

Length The length of the buffer containing the print-job title. This includes the terminating null character but excludes the Length field.

Address of Job Title The 16:16 address of the ASCIIZ string containing the print-job title. A null character terminates the ASCIIZ string.

Returns: The error return codes for this function are as follows:

0100H Completed successfully

8113H Invalid parameter, if the address of data packet is invalid, or the address of the ASCIIZ string is invalid.

Remarks: If the ASCII string has more than 126 bytes, the physical Parallel Port device driver truncates it to the first 125 bytes plus 1 byte for the null character.

Function 4EH – Set Parallel Port IRQ Time-Out Value

This function sets the parallel port IRQ timeout value.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Timeout Value in Seconds	WORD

Timeout Value The length of time, in seconds, that the physical Parallel Port device driver waits for a hardware interrupt to occur.

Returns: The error return codes for this function are as follows:

0100H Completed successfully

8113H Invalid parameter, if the data-packet address is invalid.

Remarks: This IOCTL informs the physical Parallel Port device driver of the length of time, in seconds, to wait for a hardware interrupt before a print request is canceled. Valid timeout values range between 0 and 65535 seconds. An application setting this value must also issue PrfWriteProfileString to update the OS2SYS.INI file.

Function 62H – Query Frame Control

This function returns frame control.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Characters per Line	BYTE
Lines per Inch	BYTE

Characters per Line On return, field is set to 80 or 132.

Lines per Inch On return, field is set to 6 or 8.

Returns: The error return codes for this function are as follows:

0100H Completed successfully.

Function 64H – Query Infinite Retry

This function returns infinite retry.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Data	BYTE

Data On return, Data byte is set to:

- 0 Infinite retry is disabled
- 1 Infinite retry is enabled.

Returns: The error return codes for this function are as follows:

0100H Completed successfully.

Function 66H – Query Parallel Port Status

This function returns the parallel port status.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Data	BYTE

Data On return, Data byte is set to:

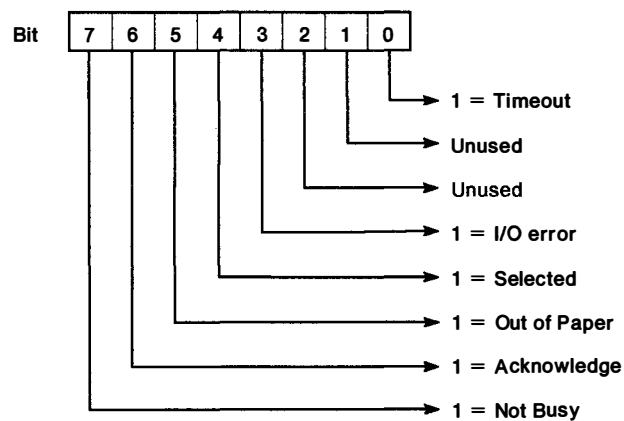


Figure 18-1. Data Byte

Returns: The error return codes for this function are as follows:

0100H Completed successfully.

Function 69H – Query Active Font

This function queries an active font.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Code Page	WORD
Font ID	WORD

Code Page On return, is set to currently active code page.

- 0000H** If the Code Page value and Font ID are returned as zero, the printer is set to the hardware default code page and font.
- 0001H – FFFFH** Valid code page numbers.

Font ID On return, is the ID value of the font that is currently active.

- 0000H** If the Code Page value and Font ID are specified as zero, the printer is set to the hardware default code page and font. If Font ID is zero and code page is non-zero, no error is returned, if any Font ID is available for the specified code page.
- 0001H – FFFFH** Valid Font ID numbers; font types defined by the font file definitions for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label, and are also entered in the DEVINFO statement for the printer.

Returns: The error return codes for this function are as follows:

- 0100H** Completed successfully
- C103H** No code page function because spooler not started
- C109H** Error caused by switcher error, not by input parameters
- C10AH** Error caused by invalid printer name as input
- C10DH** Received code page request when code page switcher not initialized
- C110H** Received request for SFN, not in SFN table.

Function 6AH – Verify Font

This function verifies that a particular code page and font is available for the specified printer.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Code Page	WORD
Font ID	WORD

Code Page The Code Page number to validate. Values can be 0–65535.

Font ID The Font ID value to validate. Values can be 0–65535. The Font ID is contained in the font file for downloadable fonts. For cartridge fonts, Font IDs are the numbers on the cartridge label, and are entered in the DEVINFO statement for the printer.

Note: A value of 0 for both the Code Page number and Font ID indicates the default hardware code page and font. This combination is always valid.

Returns: The error return codes for this function are as follows:

- C100H Completed successfully
- C102H Code page is not available
- C103H No code page function because spooler not started
- C104H Font ID is not available (verify)
- C10AH Error caused by invalid printer name as input
- C10DH Received code page request when code page switcher not initialized.

Function 6EH – Query Parallel Port IRQ Time-Out Value

This function returns a parallel port IRQ timeout value.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Timeout Value in Seconds	WORD

Timeout Value The length of time, in seconds, that the physical Parallel Port device driver waits for a hardware interrupt to occur.

Returns: The error return codes for this function are as follows:

0100H Completed successfully
 8113H Invalid parameter, if the data-packet address is invalid.

Remarks: Valid timeout values range between 0–65535 seconds.

Category 7 Mouse Control IOCTL Commands

The following is a summary of Category 7 IOCTL Commands:

Function	Description
50H	Reserved
51H	Notification of Display Mode Change
52H	Reserved
53H	Reassign Current Mouse Scaling Factors
54H	Assign New Mouse Event Mask
55H	Reserved
56H	Set Pointer Shape
57H	Unmark Collision Area
58H	Mark Collision Area
59H	Specify/Replace Pointer Screen Position
5AH	Set OS/2 Mode Pointer Draw Device Driver Address
5BH	Reserved
5CH	Set Current Physical Mouse Device Driver Status Flags
5DH	Notification of Mode Switch Completion
60H	Query Number of Mouse Buttons Supported
61H	Query Mouse Device Motion Sensitivity
62H	Query Current Physical Mouse Device Driver Status Flags
63H	Read Mouse Event Queue
64H	Query Current Event Queue Status
65H	Query Current Mouse Event Mask
66H	Query Current Mouse Scaling Factors
67H	Query Current Pointer Screen Position
68H	Query Current Pointer Shape
69H	Reserved
6AH	Query Physical Mouse Device Driver Level/Version
6BH	Query Pointing Device ID.

Function 51H – Notification of Display Mode Change

This function performs a notification of display mode change.

Parameter Packet Format: The Parameter Packet is a location in application storage that contains the Mode Data Definition record, which has the following format:

Field	Length
Length	WORD
Type	BYTE
Color	BYTE
Text Columns	WORD
Text Rows	WORD
Horizontal Resolution	WORD
Vertical Resolution	WORD
Attribute Format	BYTE
Buffer Address	DWORD
Buffer Length	DWORD
Full Buffer Size	DWORD
Partial Buffer Size	DWORD
Address of Extended Data Area	DWORD

Length

An input parameter to VioSetMode. Length specifies the length of the data structure in bytes including Length itself. The minimum structure size required is 3 bytes; the maximum structure size required is 34 bytes. OS/2 2.0 sets the first mode (in the list of modes supported by this display configuration) with a data structure matching the mode data specified.

Type

A bit mask that contains specifications for the mode being set. The definitions of the bits follow:

```

xxxxxxb b = 0 Monochrome compatible mode
          b = 1 Other
xxxxxxb b = 0 Text mode
          b = 1 Graphics mode
xxxxbxx b = 0 Enable color burst
          b = 1 Disable color burst
xxxxbxxx b = 0 VGA-compatible modes 0 - 13H
          b = 1 Native mode
bbbbxxx b = Reserved, must be 0.

```

Color

Defines the number of colors as a power of 2. This is equivalent to the number of color bits, which define the color. For example:

```

Color = 1 Specifies 2 colors
Color = 2 Specifies 4 colors
Color = 4 Specifies 16 colors
Color = 8 Specifies 256 colors
Color = 0 Should be specified for monochrome modes 7, 7+, and F.

```

Text Columns

Number of text columns.

Text Rows	Number of text rows. Twenty-five rows are supported for the color graphics adapter. Twenty-five and forty-three rows are supported for the VGA adapter. Twenty-five and fifty rows are supported for the IBM Personal System/2* Display Adapter.
Horizontal Resolution	The number of pel columns.
Vertical Resolution	The number of pel rows.
Attribute Format	Identifies the format of the attributes.
Number of Attributes	Identifies the number of attributes in a character cell.
Buffer Address	The 32-bit physical address of the physical display buffer for this mode (returned value).
Buffer Length	The length of the physical display buffer for this mode (returned value).
Full Buffer Size	The size of the buffer required for a full save of the physical display buffer for this mode (returned value).
Partial Buffer Size	The size of the buffer required for a partial (pop-up) save of the physical display buffer for this mode (returned value).
Address of Data Area	The FAR-16 address to an <i>extended-mode</i> data structure, or zero, if none. The format of the extended-mode data structure is determined by the physical device driver and is unknown to the operating system.

Data Packet Format: The Data Packet is a location in application storage that contains the configuration data for the display on which the mode is being set. The configuration data has the following format:

Field	Length
Length	WORD
Adapter Type	WORD
Display Type	WORD
Memory	DWORD
Configuration Number	WORD
Device Driver Version Number	WORD
Flag Bits	WORD
Hardware State Buffer Size	DWORD
Maximum Buffer Size - Full Save	DWORD
Maximum Buffer Size - Partial Save	DWORD
Offset to Mode Data	WORD

Length An input parameter to VioGetConfig. Length specifies the length of the data structure in bytes including Length itself. The maximum size structure required is determined by issuing VioGetConfig with Length set to 2. When Length is set to 2 on input, VioGetConfig returns the maximum size structure required in the Length field on output. When Length is not equal to 2 on input, the Length field is unmodified on output.

* Trademark of the IBM Corporation

Adapter Type

The display adapter type, with values set as follows:

- 0 = Monochrome-compatible
- 1 = Color graphics adapter
- 2 = Enhanced graphics adapter
- 3 = VGA or PS/2* display adapter
- 4-6 = Reserved
- 7 = PS/2 Display Adapter 8514/A.

Note: Values ranging from 0–4095 are reserved for IBM.

Display Type

Type of display/monitor with values indicating the following:

- 0 = Monochrome display
- 1 = Color display
- 2 = Enhanced color display
- 3 = 8503 monochrome display
- 4 = 8512 or 8513 color display
- 5-8 = Reserved
- 9 = 8514 color display.

Note: Values ranging from 0–4095 are reserved for IBM.

Memory

The amount of memory on the adapter in bytes, returned as a 32-bit value.

Configuration Number

The number of the display configuration, which this data corresponds to.

Device Driver Version #

The video device driver version number corresponding to this device driver.

Flag Bits

Defined as follows:

- 0001H** Power up display configuration
- 0002H** VGA pass-through.

Hardware State Size

The size of the buffer required by the video device driver to save the full hardware state excluding the physical display buffer.

Maximum Size - Full Save

The maximum size buffer required by the video device driver to save the full physical display buffer.

Maximum Size - Partial Save

The maximum size buffer required by the video device driver to save the portions of the physical display buffer that are overlaid by a pop-up.

Offset to Mode Data

The offset within the configuration data structure to the beginning of the mode data.

Returns: None.

Remarks: This function notifies the physical Mouse device driver of a new display mode and display configuration. When the Video Subsystem or registered Video Subsystem sets or resets the display mode, they must synchronize the physical Mouse device driver pointer update routines by providing this notification record to the physical Mouse device driver prior to switching display modes.

* Trademark of the IBM Corporation

Function 53H – Reassign Mouse Scaling Factors

This function reassigns the current mouse scaling factors.

Parameter Packet Format: The Parameter Packet is a location in application storage that contains the following data:

Field	Length
Row Data	WORD
Column Data	WORD

Row Data Row-coordinate scaling factor.

Column Data Column-coordinate scaling factor.

Scaling Factor values are positive integers in the range of:

$$0 < \text{value} \leq (32K-1)$$

Data Packet Format: None.

Returns: None.

Remarks: This function reassigns the current pointing device scaling factors. *Scaling factors* are ratio values that determine how much relative movement is necessary, before the physical Mouse device driver reports a mouse event. These ratios specify the number of mickeys per 8 pixels. The default ratio values are:

Vertical/Row ratio — 16 mickeys per 8 pixels
Horizontal/Row ratio — 8 mickeys per 8 pixels.

Function 54H – Assign New Mouse Event Mask

This function assigns a new mouse event mask.

Parameter Packet Format: The Parameter Packet is a location in application storage that contains the following data:

Field	Length
Event Mask	WORD

Event Mask Has the following bits:

- Bits 7-15** Reserved = 0
- Bit 6** Set, if Button 3 is down
- Bit 5** Set, if motion with Button 3 down
- Bit 4** Set, if Button 2 is down
- Bit 3** Set, if motion with Button 2 down
- Bit 2** Set, if Button 1 is down
- Bit 1** Set, if motion with Button 1 down
- Bit 0** Set, if all mouse motion, no buttons.

A set bit has a value of 1.

Data Packet Format: None.

Returns: None.

Remarks: The physical Mouse device driver gets the new mask for enabled/disabled mouse device events from the caller's Parameter Packet. This mask determines which events are to be queued, that is, to be read by calling "Function 63H – Read Mouse Event Queue" on page 18-136.

Function 56H – Set Pointer Shape

This function sets the pointer shape.

Parameter Packet Format: The Parameter Packet is a location in application storage that contains the Pointer Definition record. The Pointer Definition record has the following format:

Field	Length
Buffer Length	WORD
Columns	WORD
Rows	WORD
Column Hot Spot	WORD
Row Hot Spot	WORD

- Buffer Length** The length of pointer image buffer.
- Columns** The width, in columns, of pointer image.
- Rows** The height, in rows, of pointer image.
- Column Hot Spot** The column offset (within pointer image) to hot spot.
- Row Hot Spot** The row offset (within pointer image) to hot spot.

Data Packet Format: The Data Packet is a location in application storage that contains the Pointer Image buffer. The Pointer Image buffer contains a new pointer shape defined by the user for the physical Mouse device driver. The format of this buffer is dependent on the display mode. The buffer always consists of the *AND* pointer image data followed by the *XOR* pointer image data. The buffer always describes only one display plane.

Returns: None.

Remarks: In the Parameter Packet, the caller specifies a 1-WORD height value (Rows) for the pointer shape, and a 1-WORD width value (Columns) for the pointer shape. For text-mode and mono-mode applications, these values must be equal to 1. For graphics-mode applications, these values must be greater than or equal to 1.

The Length, in bytes, of the pointer shape varies, depending on the display mode:

Mono & Text

$$\begin{aligned} \text{Buffer length} &= (\text{height in characters}) * (\text{width in characters}) * 2 * 2 \\ &= 1 * 1 * 2 * 2 \end{aligned}$$

Notice that for text mode, height and width must be 1, so length is always 4.

Graphics

$$\text{Buffer length} = (\text{height in pels}) * (\text{width in pels}) * (\text{bits per pel}) * 2 / 8$$

Notice that width must be a multiple of 8.

Modes 4 & 5 (320 x 200)

$$\text{Buffer length} = (\text{height}) * (\text{width}) * 2 * 2 / 8$$

Mode 6 (640 x 200)

$$\text{Buffer length} = (\text{height}) * (\text{width}) * 1 * 2 / 8$$

Notice that length calculations produce byte boundary buffer sizes.

All of the Pointer Definition record fields and the Pointer Shape buffer are validated using the session's mode table values. The parameter values must be the same orientation as the current session display mode:

- Graphics = pixel values
- Text = character values.

Function 57H – Unmark Collision Area

This function frees the mouse to draw the pointer anywhere on the screen.

Parameter Packet Format: None.

Data Packet Format: None.

Returns: None.

Remarks: A *collision area* is a restricted area on the screen that cannot be overwritten by pointer drawing. See “Function 58H – Mark Collision Area” on page 18-127.

Function 57H frees the current collision area for a session so that the pointer can be drawn anywhere on the screen. If a collision area was declared for the session, the collision area is freed and the pointer position is checked. If the pointer was in the freed area, it is drawn. If the pointer was not in the freed area, no pointer manipulations take place.

Function 58H – Mark Collision Area

This function restricts the mouse from pointer drawing in specified areas of the screen.

Parameter Packet Format: The Parameter Packet is a location in application storage that contains a data structure that specifies a restricted area on the screen (collision area). The format of this data structure follows:

Field	Length
Left Row Position	WORD
Left Column Position	WORD
Right Row Position	WORD
Right Column Position	WORD

Left Row Position Vertical starting point of the collision area. Valid values are 0–(vertical resolution – 1).

Left Column Position Horizontal starting point of the collision area. Valid values are 0–(horizontal resolution – 1).

Right Row Position Vertical ending point of the collision area. Valid values are 0–(vertical resolution – 1).

Right Column Position Horizontal ending point of the collision area. Valid values are 0–(horizontal resolution – 1).

Data Packet Format: None.

Returns: None.

Remarks: A *collision area* is a restricted area on the screen that cannot be overwritten by pointer drawing. Values contained in the data structure defining a collision area must be specified in either character or pixel values, depending on the current display mode. If the collision area is defined as the entire screen, pointer drawing is disabled for the session.

The pointer is not drawn in this area until a different area is specified through another call to this function, or until “Function 57H – Unmark Collision Area” on page 18-126 is called.

Function 59H – Specify/Replace Pointer Position

This function specifies and replaces the pointer position.

Parameter Packet Format: The Parameter Packet is a location in application storage that contains the following data:

Field	Length
Row Position	WORD
Column Position	WORD

Row Position The new row-coordinate pointer screen position.

Column Position The new column-coordinate pointer screen position.

Data Packet Format: None.

Returns: None.

Remarks: The coordinate values are dependent on the display mode. Pixel values must be used, if the display is in graphics mode. Character position values must be used, if the display is in text mode.

This function does not override “Function 57H – Unmark Collision Area” on page 18-126, or “Function 58H – Mark Collision Area” on page 18-127, as it has no effect on current restricted areas of the screen. If the pointer is directed into a restricted area (collision area), it remains invisible until moved out of the area or until the restrictions are removed.

Function 5AH – Set OS/2 Mode Pointer Draw Device Driver Address

This function specifies the address of the physical Pointer Draw device driver.

Parameter Packet Format: The Parameter Packet is a location in application storage that contains the following data:

Field	Length
Pointer Entry	DWORD
Pointer DS	DWORD

Pointer Entry Contains two 1-WORD fields whose format is as follows:

- WORD 0** Pointer Draw device driver entry point offset
- WORD 1** Pointer Draw device driver entry point selector.

Pointer DS Contains two 1-WORD fields whose format is as follows:

- WORD 0** Pointer Draw device driver data segment selector
- WORD 1** Reserved = 0.

Data Packet Format: The Data Packet is a location in application storage that contains the following data:

Field	Length
Length of Data	WORD
Display Configuration Number	WORD
Caller	WORD

Length of Data The length of the data structure is equal to 6 bytes.

Display Configuration # The display configuration number on which the pointer draw routine should draw.

Caller Specifies whether this call is made for an application, or the Base Video Subsystem (BVS), where:

- 0** Application
- 1** BVS.

Returns: None.

Remarks: This IOCTL is for OS/2 mode only. The function passes to the physical Mouse device driver the selector:offset address of the entry point of the session's pointer draw routine for OS/2-mode display support. The pointer image draw routine is an installed *pseudo* Character device driver. The mouse router/handler must:

- Open the physical Pointer Draw device driver
- Query the physical Pointer Draw device driver for the address of its entry point
- Pass the resulting address of the pointer draw entry point to the physical Mouse device driver using the IOCTL described above.

The physical Mouse device driver issues a FAR-16 call to the physical Pointer Draw device driver when a mouse interrupt occurs that requires action concerning the pointer image.

Category 7 – Mouse IOctls

In addition, the physical Mouse device driver can call the pointer draw routine as a result of some action on the part of the application, such as:

- MouDrawPtr
- MouRemovePtr
- MouSetPtrPos
- MouSetPtrShape
- MouGetPtrShape
- MouSetDevStatus.

Function 5CH – Set Physical Mouse Device Driver Status Flags

This function sets a subset of the current Physical Mouse device driver Status flags.

Parameter Packet Format: The Parameter Packet is a location in application storage that contains the following data:

Field	Length
Status Flag	WORD

Status Flag Has the following bit level definitions:

High Byte Bit settings are as follows:

Bits 7-2 Reserved=0

Bit 1 Set, if mouse data returned in mickeys, not display units

Bit 0 Set, if the interrupt level pointer draw routine is not called.

Low Byte Bit settings are as follows:

Bits 7-0 Reserved=0

A set bit has a value of 1.

Data Packet Format: None.

Returns: None.

Remarks: This function is the complement to “Function 62H – Query Physical Mouse Driver Status Flags” on page 18-135. Only the high byte of the 1-WORD physical Mouse device driver Status Flag can be set. If the Data Format flag is altered, the monitor chain and event queue are flushed for the calling session of this IOctl.

It is useful to disable interrupt-time Pointer Draw device driver activity when an application is going to use an unsupported display mode for which it intends to provide the pointer image drawing support. If this is desired, the application must set this mouse state prior to switching the display mode.

Function 5DH – Notification of Mode Switch Completion

This function informs the mouse that a display mode or configuration switch has been completed, and that drawing operations can resume.

Parameter Packet Format

Field	Length
Dummy FAR-16 Pointer (such as a doubleword of zeroes)	DWORD

Data Packet Format

Field	Length
Dummy FAR-16 Pointer (such as a doubleword of zeroes)	DWORD

Returns: None.

Remarks: This call does not provide return parameters or support input parameters. If the pointer was hidden on the mode switch start (Function 51H), then it is redrawn.

This function is the complement to “Function 51H – Notification of Display Mode Change” on page 18-119.

Function 60H – Query Number of Mouse Buttons Supported

This function returns the number of buttons supported by the physical Mouse device driver.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a location in application storage where the physical Mouse device driver returns to the caller the number of buttons supported by the physical device driver. The format of the Data Packet is as follows:

Field	Length
Number Supported	WORD

Number Supported The number of buttons supported by the physical Mouse device driver. Return values are in the range of:

- 1 One-button support
- 2 Two-button support
- 3 Three-button support.

Returns: This function returns to the caller the number of buttons supported by the physical Mouse device driver.

Function 61H – Query Mouse Device Motion Sensitivity

This function returns the mouse device's motion sensitivity.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a location in application storage where the physical Mouse device driver returns the mouse device's motion sensitivity to the caller. The format of the Data Packet is as follows:

Field	Length
Mickeys/Centimeter	WORD

Mickeys/Centimeter The number of mickeys/centimeter supported by the mouse device. Return values are in the range of:

$$0 < \text{number of mickeys/centimeter} \leq (32K - 1)$$

Returns: This function returns to the caller the motion sensitivity of the mouse device as represented by the number of mickeys/centimeter.

Function 62H – Query Physical Mouse Driver Status Flags

This function returns the current physical Mouse device driver Status flags.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a location in application storage where the physical Mouse device driver returns the current device Status flags to the caller. The format of the Data Packet is as follows:

Field	Length
Status Flags	WORD

Status Flags Has the following bit level definitions:

High Byte Bit settings are as follows:

Bits 7-2 Reserved=0

Bit 1 Set, if mouse data returned in mickeys

Bit 0 Set, if the interrupt level pointer draw routine is not called.

Low Byte Bit settings are as follows:

Bits 7-4 Reserved=0

Bit 3 Set, if pointer draw routine disabled by unsupported mode

Bit 2 Set, if flush in progress

Bit 1 Set, if block read in progress

Bit 0 Set, if event queue busy with I/O.

A set bit is a value of 1.

Returns: This function returns to the caller the current physical Mouse device driver Status flags.

Remarks: This function is the complement to "Function 5CH – Set Physical Mouse Device Driver Status Flags" on page 18-131.

Function 63H – Read Mouse Event Queue

This function reads the mouse event queue.

Parameter Packet Format

Field	Length
Read Type	WORD

Read Type Used only to determine the type of action taken, if no event queue data is available. The values are:

- 0 Block the process (Wait) until event data is available
- 1 Return a NULL record (No-Wait) for the request.

Data Packet Format: The Data Packet is a location in application storage where the Mouse device driver returns to the caller a 10-byte mouse event queue record. A mouse event queue record has the following format:

Field	Length
Event Mask	WORD
Time	DWORD
Row Position	WORD
Column Position	WORD

Event Mask See “Function 65H – Query Mouse Event Mask” on page 18-138

Time Event time stamp in milliseconds

Row Position Row-coordinate pointer

Column Position Column-coordinate pointer.

Returns: This function returns to the caller a mouse event from the mouse event (FIFO) queue.

Function 64H – Query Event Queue Status

This function returns the current event queue status.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a location in application storage where the physical Mouse device driver returns to the caller the following data:

Field	Length
Element Count	WORD
Queue Size	WORD

Element Count The current number of event queue elements. The return value is a 1-WORD value in the range of:

$$0 \leq \text{value} \leq \text{MaxNumQueueElements}$$

Queue Size A 1-WORD return value for the MaxNumQueueElements.

Returns: This function returns to the caller the current number of queued elements in the mouse event queue, and the maximum number of elements allowed in a mouse event queue.

Remarks: MaxNumQueueElements is derived from the CONFIG.SYS QSIZE = keyword input parameter.

Function 65H – Query Mouse Event Mask

This function returns the current mouse event mask.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a location in application storage where the physical Mouse device driver returns to the caller a 1-WORD current mouse Event Mask. The format of the Data Packet is as follows:

Field	Length
Event Mask	WORD

Event Mask Has the following bit level definitions:

- Bits 7-15** Reserved. Must be 0.
- Bit 6** Set, if Button 3 is down.
- Bit 5** Set, if motion, with Button 3 down.
- Bit 4** Set, if Button 2 is down.
- Bit 3** Set, if motion, with Button 2 down.
- Bit 2** Set, if Button 1 is down.
- Bit 1** Set, if motion, with Button 1 down.
- Bit 0** Set, if all mouse motion, no buttons.

A set bit has a value of 1.

Returns: This function returns to the caller the current mouse event mask.

Remarks: The return values may be any valid combination of enabled/disabled event flags. This is the complement function for “Function 54H – Assign New Mouse Event Mask” on page 18-123.

Function 66H – Query Mouse Scaling Factors

This function returns the current mouse scaling factors.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a location in application storage where the physical Mouse device driver returns to the caller the current mouse scaling factors. The format of the Data Packet is as follows:

Field	Length
Row Data	WORD
Column Data	WORD

Row Data The row-coordinate scaling factor.

Column Data The column-coordinate scaling factor. The scaling factor values are positive integers in the range of:

$$0 < \text{value} \leq (32K-1)$$

Returns: This function returns to the caller the current mouse scaling factors.

Remarks: Scaling factors are ratio values that determine how much relative movement is necessary before the physical Mouse device driver reports a mouse event. The ratios specify the number of mickeys per 8 pixels. The default ratio values are:

Vertical/Row ratio – 16 mickeys per 8 pixels

Horizontal/Row ratio – 8 mickeys per 8 pixels.

This is the complement function for “Function 53H – Reassign Mouse Scaling Factors” on page 18-122.

Function 67H – Query Pointer Screen Position

This function returns the current pointer screen position.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a location in application storage where the physical Mouse device driver returns to the caller the current pointer screen position. The format of the Data Packet follows:

Field	Length
Row Position	WORD
Column Position	WORD

Row Position The row-coordinate pointer screen position.

Column Position The column-coordinate pointer screen position.

Returns: This function returns to the caller the current pointer screen position.

Remarks: The coordinate values are display-mode dependent. Pixel values are returned if the display is in graphics mode. Character position values are returned if the display is in text mode.

Function 68H – Query Pointer Shape

This function returns the current pointer shape.

Parameter Packet Format: The Parameter Packet is a location in application storage containing the Pointer Definition record, which describes a user buffer where the physical Mouse device driver returns the current pointer shape. This data has the following format:

Field	Length
Buffer Length	WORD
Columns	WORD
Rows	WORD
Column Hot Spot	WORD
Row Hot Spot	WORD

- Buffer Length** The length of the (user) Pointer Shape buffer in bytes. When calling this function, the caller must specify the length of the Pointer Shape buffer in this field.
- Columns** The width, in columns, of the pointer image.
- Rows** The height, in rows, of the pointer image.
- Column Hot Spot** The column offset within the pointer image to the hot spot.
- Row Hot Spot** The row offset within the pointer image to the hot spot.

Data Packet Format: The Data Packet is a location in application storage where the physical Mouse device driver returns the current pointer shape, a user buffer described by the data in the Parameter Packet.

Returns: If the Buffer Length value specified by the caller is smaller than the required storage for the pointer shape, the physical Mouse device driver returns an error. The physical Mouse device driver returns the minimum storage requirements for the pointer shape in the Buffer Length field.

If the Buffer Length value specified by the caller is greater than, or equal to, the amount of storage required for the pointer shape, the physical Mouse device driver returns the pointer shape in the user's Pointer Shape buffer (that is, in the Data Packet) and the data describing the pointer shape in the Pointer Definition record (Parameter Packet). The actual returned length is returned in the Buffer Length field.

Remarks: The Pointer Shape buffer is described by the Pointer Definition record, and for normal conditions, consists of the Screen AND, and Pointer XOR, masks.

Function 6AH – Query Physical Mouse Device Driver Version Number

This function returns the physical Mouse device driver level/version number.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a WORD in application storage, where the physical Mouse device driver returns its version number.

Field	Length
Version Number	WORD

Version Number Has the following settings:

- 1 OS/2 1.3 level support for device-dependent device drivers
- 2 OS/2 2.0 level support for device-dependent device drivers.

Returns: None.

Function 6BH – Query Pointing Device ID

This function returns the Pointing Device ID.

Parameter Packet Format: None.

Data Packet Format: The Data Packet is a location in application storage with the following format:

Field	Length
Device ID	WORD
COM Port Number	WORD
Secondary Device ID	WORD

Device ID The ID number for the primary pointing device.

COM Port Number The number of the communications port connected to the serial pointing device.
Valid COM Port Numbers are:

- 1 Communications Port 1
- 2 Communications Port 2
- 3 Communications Port 3
- 4 Communications Port 4.

Secondary Device ID The Pointing Device ID for the secondary device.

Returns: This function fills the caller's Data Packet fields.

Remarks: Device IDs are specific to the type of pointing device installed. The supported Pointing Device IDs are listed below:

- 0 Unknown
- 1 Bus mouse
- 2 Serial mouse
- 3 Inport mouse
- 4 PS/2-style pointing device
- 5 IBM 8516 Touch Display
- 6–65535 Reserved.

Note: If a serial pointing device is not installed, the COM Port Number field is meaningless.

Category 8 Logical Disk Control IOCTL Commands

The following is a summary of Category 8 IOCTL Commands:

Function	Description
00H	Lock Drive
01H	Unlock Drive
02H	Redetermine Media (end format)
03H	Set Logical Map
04H	Begin Format
20H	Block Removable
21H	Query Logical Map
22H	Reserved
43H	Set Device Parameters
44H	Write Track
45H	Format and Verify Track
5EH	Reserved
5FH	Reserved
60H	Query Media Sense
63H	Query Device Parameters
64H	Read Track
65H	Verify Track

Function 00H – Lock Drive

This function locks a drive and is used to exclude I/O by another process on the volume in the drive. The Lock Drive function succeeds only if there is one file handle open on the volume in the drive; that is, the file handle on which this function is issued. This is necessary since the desired result is to exclude all other I/O to this volume, until "Function 01H – Unlock Drive" is issued.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: This function should be called after obtaining a handle from DosOpen but before actually accessing the drive. Notice that it is necessary to call "Function 01H – Unlock Drive" on page 18-146 before closing the drive.

Function 01H – Unlock Drive

This function unlocks the drive.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: This function releases a lock on a volume in a driver, thereby allowing I/O from other processes to that volume again. The locked volume, represented by the file handle on which this function is issued must be in the drive.

Notice that this function must be called after accessing a drive that was locked with “Function 00H – Lock Drive” on page 18-145, before closing the drive with DosClose.

Function 02H – Redetermine Media

This function redetermines media (ends format).

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: This function rebuilds the device parameters (including the Volume Parameter Block (VPB) used by the operating system to identify the volume), simulates a close on the current device handle, and forces a mount on the volume.

Function 02H dismounts the volume from the current FSD, attaches the new FSD, reattaches the current handle to the volume's new FSD, and is typically called after a new boot sector has been written to a volume (after a format has been done). The caller must have the disk or diskette locked when calling this function, otherwise, the function fails with the error ERROR_LOCK_VIOLATION.

The caller can have only one file open to refer to the disk or diskette. If other processes have the volume open, or the calling process has opened the volume multiple times, the call fails and ERROR_DRIVE_BUSY is returned.

Function 03H – Set Logical Map

This function sets the next logical drive letter that is used to reference the drive.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Logical Drive Number	BYTE

Logical Drive Number On entry, a logical drive number (1 = A, 2 = B, and so forth) is specified. On return, this byte specifies the logical drive currently mapped to the drive that the specified file handle is opened on. A zero is returned if there is only one logical drive mapped onto this physical drive.

Returns: See the DosDevIOCtrl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: When copying diskettes on a drive whose physical drive number has more than one logical drive letter assigned to it, the operating system issues *diskette swap* prompts to tell the user which logical drive letter is currently referencing the physical drive number. Function 03H is typically used to avoid this prompt by setting a drive number (corresponding to the drive letter) that is referenced in the next I/O request. The last logical drive letter assigned to the physical drive is determined by calling "Function 21H – Query Logical Map" on page 18-151.

Function 04H – Begin Format

This function begins the format.

Parameter Packet Format

Field	Length
FSD Name	ASCIIZ string

FSD Name The file system driver name, that is, the name the FSD exports. A zero length string is used to indicate the FAT file system.

Data Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Returns: See the DosDevIOctl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: This function attaches (mounts) a specified File System Driver (FSD) to a logical disk volume. Function 04H is typically used to force mount the FSD that is formatting a volume. It is called before starting a format operation on a volume. This function also unmounts a current FSD (if any), and forces a mount to the specified FSD.

A flag is set in the OS/2 kernel to indicate that a Begin Format (unmount/mount) was done. "Function 02H – Redetermine Media" on page 18-147 clears this flag. Therefore, Function 02H must be performed before the disk is closed with DosClosed. The volume is then ready for normal I/O service from the newly mounted FSD.

Function 20H – Block Removable

This function is used to determine if the media is removable or fixed.

Parameter Packet Format

Field	Length
Command Information	BYTE
Drive Unit	BYTE

Command Information Reserved. Must be set to 0.

Drive Unit This field is used only when this IOCTL is issued without using a previously allocated file handle. A file handle of -1 must be used. Notice that media in the drive is not required. Drive Unit values are 0=A, 1=B, 2=C, and so forth.

Data Packet Format

Field	Length
Data	BYTE

Data On return, the data byte is set accordingly:

- 0 Removable media
- 1 Nonremovable media.

Returns: See the DosDevIOCTL error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Function 21H – Query Logical Map

This function returns the logical drive letter that was last used to reference (open) the drive.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Logical Drive Number	BYTE

Logical Drive Number On entry, a logical drive number (1 = A, 2 = B, and so forth) is specified. On return, if the device has more than one logical drive letter assigned to it, a drive number corresponding to the last drive letter that was used to reference the device is returned. For example, if the entry drive number was 1 (where 1 = A) and the returned was 2 (where 2 = B), this drive was last referenced as the **B:** drive. If only one drive letter (logical drive) is assigned to the device, a zero is returned.

Note: This function works as long as the file handle is valid. The file handle can be set to anything other than zero.

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Function 43H – Set Device Parameters

This function sets the parameters for a specified device.

Parameter Packet Format

Field	Length
Command Information	BYTE
Drive Unit	BYTE

Command Information The two low bits of the command byte are used to indicate one of three possible actions:

Bits Description

- | | |
|----|---|
| 00 | Revert to building the BPB off the medium for all subsequent Build BPB functions. This is used after a format operation to reset the device parameters to their original state. |
| 01 | Change the default BPB for the physical device. This changes the physical parameters <i>for</i> the drive as opposed to parameters for the media <i>in</i> the drive. |
| 10 | Change the BPB for the medium to the specified BPB, and return the new BPB for the medium for all subsequent Build BPB calls. This is used to prepare the device for a format media operation according to the device parameters specified. |

All other bits are reserved, and must be set to 0.

Drive Unit Used only when this IOCTL is issued without using a previously allocated file handle, and Command Information is set to 01. A file handle of -1 must be used. Notice that media in the drive is not required. Drive Unit values are 0=A, 1=B, 2=C, and so forth.

Data Packet Format

Field	Length
Extended BPB for Devices	31 BYTES
Number of Cylinders	WORD
Device Type	BYTE
Device Attributes	WORD

Extended BPB The term *Extended BPB* refers to the BIOS Parameter Block (a table that describes the structure of the media), with a description of the the physical layout of the drive (heads, tracks, sectors).

The Extended BPB has the following format:

Field	Length
Bytes per Sector	WORD
Sectors per Cluster	BYTE
Reserved Sectors	WORD
Number of FATs	BYTE
Root Dir Entries	WORD
Total Sectors	WORD
Media Descriptor	BYTE
Sectors per FAT	WORD
Sectors per Track	WORD
Number of Heads	WORD
Hidden Sectors	DWORD
Large Total Sectors	DWORD
Reserved	6 BYTES

Number of Cylinders Indicates the number of cylinders defined for the physical device.

Device Type Describes the physical layout of the device specified, and has one of the following values:

- 0** 48 TPI low-density diskette drive
- 1** 96 TPI high-density diskette drive
- 2** Small (3.5-inch) 720KB drive
- 3** 8-inch single density floppy drive
- 4** 8-inch double density floppy drive
- 5** Fixed disk
- 6** Tape drive
- 7** Other (includes 1.44MB 3.5-inch diskette drive)
- 8** R/W optical disk
- 9** 3.5-inch 4.0MB diskette drive (2.88MB formatted)

Device Attributes A bit field that returns various flag information about the specified drive:

- Bit 0** Removable Media flag. This bit is set to 1, if the media cannot be removed. It is set to 0, if the media is removable.
- Bit 1** Changeline flag. This bit is set to 1, if the device support determines that the media was removed since the last I/O operation. To query whether the media has changed, call the device driver Strategy Command "1H / MEDIA CHECK" on page 15-7.

If this bit is set to 0, then the physical device driver can return the value 0, *Unsure if media has changed*, from the Media Check function.

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Functions 44H, 64H, 65H – Write/Read/Verify Track

These functions write, read, and verify a track (write from 44H, read to 64H, and verify track 65H).

Parameter Packet Format: The three functions above have the same Parameter Packet as shown below:

Field	Length
Command Information	BYTE
Head	WORD
Cylinder	WORD
First Sector	WORD
Number of Sectors	WORD
Track Layout Table	BYTES

Command Information A byte with bit 0 defined as follows:

Bit 0 If clear (0), track layout contains non-consecutive sectors or does not start with Sector 1. If set (1), track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

Head The physical head on the drive that performs the operation.

Cylinder The cylinder for the write/read/verify.

First Sector The logical sector number within the Track Layout Table that starts the I/O operation. Notice that the sector numbers start with 0. For example, the third sector is number 2.

Number of Sectors The number of sectors to write/read/verify (up to the maximum specified in the track table. The IOCTL subfunctions do not step heads/tracks).

Track Layout Table Has the following:

Field	Length
Sector Number for Sector 1	WORD
Sector Size for Sector 1	WORD
Sector Number for Sector 2	WORD
Sector Size for Sector 2	WORD
Sector Number for Sector 3	WORD
Sector Size for Sector 3	WORD
•	•
•	•
•	•
Sector number for Sector <i>n</i>	WORD
Sector size for Sector <i>n</i>	WORD

The sector table that is specified provides information, which is used during the WRITE/READ/VERIFY track operations.

Data Packet Format: The Data Packet is a buffer. For WRITE Function 44H, it contains the data to be written. For READ Function 64H, the buffer must be large enough to hold requested data. For VERIFY Function 65H, the Data Packet is not used.

Field	Length
Buffer	BYTES

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: These functions are used to perform a write, read, or verify of sectors on the media, and to perform the operations on the device that is specified in this request. The track table passed in on the call is used to determine the sector number, which is passed to the disk controller for the operation. In cases where the sectors are oddly numbered, or are non-consecutive, this request breaks into *n* single sector operations and reads/writes/verifies one sector at a time. Note also that the device driver does not correctly read a non-512 byte sector, if the READ operation would generate a DMA violation error. *Applications should be written so that this error does not occur.* These DosDevIOctls are typically used when performing I/O outside of the normal file system data area on the media.

Function 45H – Format and Verify Track

This function formats and verifies a track.

Parameter Packet Format

Field	Length
Command Information	BYTE
Head	WORD
Cylinder	WORD
Number of Tracks	WORD
Number of Sectors	WORD
Format Track Table	BYTES

Command Information A byte with bit 0 defined as follows:

Bit 0 If clear (0), track layout contains non-consecutive sectors or does not start with Sector 1. If set (1), track layout starts with Sector 1 and contains only consecutive sectors.

All other bits are reserved and must be set to 0.

Head The physical head on the drive to perform the operation.

Cylinder The cylinder for the operation. Upon return from a multi-track format request, if a defective spot is encountered on the media, the returned head and cylinder contain the defective area.

Number of Tracks The number of tracks to format/verify on a multi-track request. This is set to zero for single track requests. On return from a multi-track request, Number of Tracks is set to one of the following values:

Value	Description
8000H	Multi-track successful
4000H	Multi-track not supported
2000H	Multi-track failed on FORMAT operation
1000H	Multi-track failed on VERIFY operation.

Number of Sectors The number of sectors on the track being formatted. On return from a multi-track request, if a defective spot is found on the media before the multi-track format operation can complete, this field indicates the number of tracks remaining (to be formatted) on this multi-track format request.

Format Track Table Contains four-byte tuples. Each *tuple* is in the form (c, h, r, n) where c=cylinder number, h=head number, r=Sector ID, and n=save bytes per sector:

n	Bytes/Sector
0	128
1	256
2	512
3	1024

There is a 4-tuple for each sector in the track to be formatted. Both the head and cylinder numbers must be consistent within the tuple and the corresponding Parameter Packet field.

Data Packet Format

Field	Length
Starting Sector	BYTE

Starting Sector On input, this is the starting sector on a multi-track request. This is set to zero for a single track request. On return from a multi-track format request, if a defective spot is found on the media, Starting Sector is the first logical sector number within the head and cylinder of the defective area.

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: This routine formats and verifies the track specified according to the information passed in the Track Layout field. The track layout is passed to the controller, which performs the formatting. Notice that some controllers do not support formatting tracks with varying sector sizes, therefore, applications must ensure that the sector sizes specified in the Track Layout Table are all the same.

Function 60H – Query Media Sense

This function returns the media sense information.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Media Sense Information	BYTE

Media Sense Information On return, this field can be interpreted as follows:

- 0 Unable to determine media type
- 1 720KB diskette is present in 3.5-inch drive
- 2 1.44MB diskette is present in 3.5-inch drive
- 3 2.88MB diskette is present in 3.5-inch drive.

Returns: The error return codes for this function are as follows:

- 0 NO_ERROR
- 1 ERROR_INVALID_FUNCTION
- 6 ERROR_INVALID_HANDLE
- 15 ERROR_INVALID_DRIVE
- 22 ERROR_BAD_COMMAND
- 31 ERROR_GEN_FAILURE
- 87 ERROR_INVALID_PARAMETER
- 115 ERROR_PROTECTION_VIOLATION
- 117 ERROR_INVALID_CATEGORY
- 119 ERROR_BAD_DRIVER_LEVEL
- 163 ERROR_UNCERTAIN_MEDIA
- 165 ERROR_MONITORS_NOT_SUPPORTED.

Function 63H – Query Device Parameters

This function returns the device parameters.

Parameter Packet Format

Field	Length
Command Information	BYTE
Drive Unit	BYTE

Command Information A byte with bit 0 defined as follows:

- 0** Return the recommended BPB for the drive, which is the BPB for the physical device, unless it is a formatted fixed media. Then, it is the BPB that was on the media when the system was booted.
- 1** Return the BPB for the media currently in the drive. This always reads the BPB off the current media in the drive. An error is returned, if the media is unformatted.

All other bits are reserved, and must be set to 0.

Drive Unit

Used only when this IOCTL is issued without using a previously allocated file handle, and Command Information is set to 0. A file handle of -1 must be used. Notice that media in the drive is not required. Drive Unit values are 0=A, 1=B, 2=C, and so forth.

Data Packet Format

Field	Length
Extended BPB for Device	31 BYTES
Number of Cylinders	WORD
Device Type	BYTE
Device Attributes	WORD

Extended BPB

The physical device driver maintains two BPBs for each drive. One is the current BPB that corresponds to the media in the drive. The other is a recommended BPB based on the type of media that corresponds to the physical device (for example, for a high-density drive, the BPB is for a 96 tracks-per-inch TPI floppy; for a low-density drive, the BPB is for a 48 TPI floppy). The low bit of the Command Information field indicates which BPB the application needs to see.

Number of Cylinders

Indicates the number of cylinders defined for the physical device.

Device Type

Describes the physical layout of the device specified, and has one of the following values:

- 0** 48 TPI low-density diskette drive
- 1** 96 TPI high-density diskette drive
- 2** 3.5-inch 720KB diskette drive
- 3** 8-Inch single density diskette drive
- 4** 8-Inch double density diskette drive
- 5** Fixed disk
- 6** Tape drive
- 7** Other (includes 1.44MB 3.5-inch diskette drive)

- 8 R/W optical disk
- 9 3.5-inch 4.0MB diskette drive (2.88MB formatted)

Device Attributes

A bit field that returns various flag information about the specified drive:

Bit 0 Removable Media flag. This bit is set to 1, if the media cannot be removed. It is set to 0, if the media is removable.

Bit 1 Changeline flag. This bit is set to 1, if the device support determines that the media was removed since the last I/O operation. To query whether the media has changed, call the physical device driver Strategy Command "1H / MEDIA CHECK" on page 15-7.

If this bit is set to 0, then the physical device driver can return the value 0, *Unsure if media has changed*, from the Media Check function.

Bit 2 Greater Than 16MB Support flag. If this bit is set to 1, the physical device driver supports physical addresses greater than 16MB.

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: This function gets the parameters for a specified device.

Category 9 Physical Disk Control IOCTL Commands

Category 9 is used to access physical partitionable fixed disks. The handle, used for Category 9 commands, is returned by the DosPhysicalDisk (Function 2) API function (see the *OS/2 2.0 Control Program Programming Reference* for more information). This handle is used to tell the system which physical disk is accessed by the IOCTL command.

The Physical Disk Control commands relate to the entire partitionable fixed disk. Direct track and sector I/O begin at the beginning of the physical drive. "Function 63H – Query Physical Device Parameters" on page 18-166, describes the entire physical device.

The following is a summary of Category 9 IOCTL Commands:

Function	Description
00H	Lock Physical Drive
01H	Unlock Physical Drive
44H	Physical Write Track
63H	Query Physical Device Parameters
64H	Physical Read Track
65H	Physical Verify Track.

Function 00H – Lock Physical Drive

This function locks the physical drive.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: See the DosDevIOctl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: All the logical units on the physical drive are affected as well.

Function 01H – Unlock Physical Drive

This function unlocks the physical drive.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: All the logical units on the physical drive are affected as well.

Functions 44H, 64H, 65H – Write/Read/Verify

These functions perform a physical write, read, and verify track (write from 44H, read to 64H, and verify track 65H).

Parameter Packet Format: The three functions above have the same Parameter Packet shown below:

Field	Length
Command Information	BYTE
Head	WORD
Cylinder	WORD
First Sector	WORD
Number of Sectors	WORD
Track Layout Table	BYTES

Command Information A bit field as follows:

Bit 0 If clear (0), track layout contains non-consecutive sectors or does not start with Sector 1. If set (1), track layout start with Sector 1 and contains only consecutive sectors.

All other bits are reserved, and must be 0.

Head The physical head on the drive to perform the operation.

Cylinder The cylinder for the write/read/verify.

First Sector The logical sector number within the Track Layout Table to start the I/O operation. Notice that the sector numbers start with 0. For example, the third sector is numbered, 2.

Number of Sectors The number of sectors to write/read/verify (up to the maximum specified in the track table; the IOctl subfunctions do not step heads/tracks).

Track Layout Table The Track Layout Table is as follows:

Field	Length
Sector Number for Sector 1	WORD
Sector Size for Sector 1	WORD
Sector Number for Sector 2	WORD
Sector Size for Sector 2	WORD
Sector Number for Sector 3	WORD
Sector Size for Sector 3	WORD
•	•
•	•
•	•
Sector Number for Sector <i>n</i>	WORD
Sector Size for Sector <i>n</i> .	WORD

Data Packet Format: The Data Packet is a buffer. For WRITE Function 44H, it contains the data to be written. For READ Function 64H, the buffer must be large enough to hold requested data. For VERIFY Function 65H, the Data Packet is not used.

Field	Length
Buffer	BYTES

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: These functions perform the operations on the physical drive that are specified in this request. This is similar to the Category 8 commands, except that the I/O is done offset from the beginning of the physical drive instead of from the beginning of the extended volume associated with the unit number (Category 8).

The Track Layout Table passed in the Parameter Packet is used to determine the sector number, which is passed on to the disk controller for the operation. In cases where the sectors are oddly numbered, or are non-consecutive, the request is broken into *n* single sector operations and is written/read/verified one sector at a time. Notice that the device driver does not correctly read a non-512 byte sector if the READ operation would generate a DMA violation error. *Applications must ensure that this error does not occur.* The sector table that is specified provides information, which is used during the WRITE/READ/VERIFY track operations.

Function 63H – Query Physical Device Parameters

This function returns the physical device parameters.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved	WORD
Number of Cylinders	WORD
Number of Heads	WORD
Number of Sectors per Track	WORD
Reserved	WORD
Reserved	WORD
Reserved	WORD
Reserved	WORD

Number of Cylinders The number of cylinders on the physical drive.

Number of Heads The number of heads on the physical drive.

Number of Sectors The number of sectors per track on the physical drive.

Returns: See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference*.

Remarks: The data values returned apply to the entire physical disk.

Category 10 Character Device Monitor IOCTL Command

The following is the Category 10 IOCTL Command:

Function	Description
40H	Register Monitor

Function 40H – Register Monitor

This function registers a monitor.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format: The Data Packet includes the following parameters passed from a DosMonReg function:

Field	Length
Placement Flag	WORD
Index	WORD
Address of Input Buffer	DWORD
Offset of Output Buffer	WORD

Placement Flag The DosMonReg function call parameter that is used by an application to indicate:

- Where its monitor buffers are to be placed within a monitor chain relative to monitors already registered on the monitor chain
- What special processing requirements need to be supported by the monitor dispatcher.

Refer to the DosMonReg function description in the *OS/2 2.0 Control Program Programming Reference* for valid parameter values.

Index Used by an application to indicate on which monitor chain its monitor buffers are being registered. The accepted values for this parameter vary for each device driver. See Chapter 12, “Physical Mouse Device Driver,” Chapter 11, “Physical Keyboard Device Driver,” and Chapter 13, “Physical Parallel Port Device Driver” to determine the valid parameter value for each device driver.

Address of Input Buffer Specifies the address of a monitor input buffer allocated by an application, initialized by the monitor dispatcher, and used by the DosMonRead function.

Offset of Output Buffer Specifies the offset of a monitor output buffer allocated by an application in the same data segment as the input buffer, initialized by the monitor dispatcher, and used by the DosMonWrite function.

Refer to the descriptions of the DosMonReg, DosMonRead, and DosMonWrite functions in the *OS/2 2.0 Control Program Programming Reference* for more information.

Returns: An error is returned to the caller, if monitor registration fails because of invalid parameters (for example, the specification of an invalid monitor chain (Index), the monitor buffers are too small, or there are not enough system resources).

When an error condition occurs, the following return codes are found in the request packet status field:

8112H NO_MONITOR_SUPPORT
8103H BAD_COMMAND
810CH GENERAL_FAILURE.

Remarks: Character device drivers that support device monitors receive this request when an application calls DosMonReg to register a monitor. The physical device driver places the values of these parameters in registers and calls the Device Helper service, "Register" on page 17-70, to complete monitor registration.

Category 11 General Device Control IOCTL Commands

The following is a summary of Category 11 IOCTL Commands:

Function	Description
01H	Flush Input Buffer
02H	Flush Output Buffer
41H	System Notifications for Physical Device Drivers
60H	Query Monitor Support

Function 01H – Flush Input Buffer

This function flushes the input buffer.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: None.

Function 02H – Flush Output Buffer

This function flushes the output buffer.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: None.

Function 41H – System Notifications for Physical Device Drivers

This function notifies the physical device driver of any activities or modifications that occur during session switching, and of any changes to the alternate-input-methods interfaces.

Parameter Packet Format

Field	Length
Length	WORD
Action	WORD
Call Data (number of bytes varies with the value of Action)	BYTES

Length Contains the size of this structure, in bytes, including Length itself. Valid Length values are 8, 12, and 24. Length values of 8 and 12 are used for notifications pertaining to session information, such as session creation, termination, saving/restoring of sessions, and session switching. A length value of 24 is used for Alternate Input Methods (AIM) value verifications and activations. Notifications received with Length values other than those defined above should be returned with an ERROR_INVALID_PARAMETER error code.

Action Contains a value indicating the notification-calling condition. Action is a bitmap, which describes the type of notification made to the physical device driver. The bitmap definitions are as follows:

Bits 15–9 Reserved. Must equal 0.

Bit 8 Action = 256. AIM post-save activation.

Bit 7 Action = 128. AIM pre-save verification.

Bit 6 Action = 64. End of session switch.

Bit 5 Action = 32. Start of session switch.

Bit 4 Action = 16. Session creation.

Bit 3 Action = 8. Session termination.

Bit 2 Action = 4. Post-session restore.

Bit 1 Action = 2. Post-session save.

Bit 0 Action = 1. Pre-session save.

Physical device drivers can register for any combination of these notification values by issuing the DOSSMRegisterDD API function (see the *OS/2 2.0 Control Program Programming Reference* for more information). If a physical device driver receives a notification with an Action value it does not support, it returns an ERROR_INVALID_PARAMETER error code.

Call Data The combination of the Length field value and Action field values define the Call Data area size, format, and field definitions. The valid Call Data formats supported by this interface are defined as follows:

Length = 8

Action = 1. Pre-session save. Call Data contains the following fields to represent the outgoing session information:

Field	Length
Session Type	WORD
Session ID	WORD

Action = 2. Post-session save. Call Data contains the following fields to represent the changing (incoming and outgoing Session IDs) session information:

Field	Length
Session ID In	WORD
Session ID Out	WORD

Action = 4. Post-session restore. Call Data contains the following fields to represent the restored (incoming) session information:

Field	Length
Session ID	WORD
Session Type	WORD

Action = 8. Session termination. Call Data contains the following fields to represent the terminating session information:

Field	Length
Session Type	WORD
Session ID	WORD

Action = 16. Session creation. Call Data contains the following fields to represent the newly created session information:

Field	Length
Session ID	WORD
Session Type	WORD

Length = 12

Action = 32. Pre-session switch. Call Data contains the following fields to represent session-switching (incoming and outgoing session) information:

Field	Length
Session In ID	WORD
Session In Type	WORD
Session Out ID	WORD
Session Out Type	WORD

Action = 64. Post-session switch. Call Data contains the following fields to represent session-switching information:

Field	Length
Session In ID	WORD
Session In Type	WORD
Session Out ID	WORD
Session Out Type	WORD

Length = 24

Action = 128. AIM pre-save verify. This type of notification is issued to device drivers to verify that the new AIM parameters are valid. The new values are not actually set until the AIM post-save activate notification.

Action = 256. AIM post-save activate. The Call Data area format for Length = 24 (all Action values), is defined as follows:

Field	Length
AIM_Errors	DWORD
AIM_Active	WORD
AIM_Timeout	WORD
AIM_FKAccept	DWORD
AIM_FKRate	DWORD
AIM_FKDelay	DWORD

AIM_Errors

Contains a return bit mask, which identifies the specific AIM parameter value/field that caused an error condition. A set bit (value of 1) indicates an error. The AIM_Errors bit definitions are as follows:

Bits 31 – 5 Reserved. Must equal 0.

Bit 4 Set, if AIM_FKDelay value error.

Bit 3 Set, if AIM_FKRate value error.

Bit 2 Set, if AIM_FKAccept value error.

Bit 1 Set, if AIM_Timeout value error.

Bit 0 Set, if AIM_Active value error.

Note: Device drivers performing AIM parameter validation must do so on the pre-save notification calls. Error return codes are not meaningful, and are not returned on AIM post-save notification calls.

AIM_Active

Contains the Special Needs Status flag. Possible AIM_Active values are:

0 Indicates that AIM Support is currently inactive (FALSE).

1 Indicates that AIM Support is currently active (TRUE).

AIM_Timeout

Contains the number of seconds that the Special Needs methods should remain active, once keyboard usage has ceased. A value of 0 indicates no timeout.

AIM_FKAccept	Contains the number of milliseconds a key must be pressed in order for it to be recognized as a <i>key press</i> event. A value of 0 indicates that no additional time is applied to the standard hardware response time
AIM_FKRate	Indicates the number of milliseconds between recognized typematic key press events when a key is held down. A value of 0 indicates that no additional time is applied to the standard hardware rate.
AIM_FKDelay	Contains the number of milliseconds that a key press must be held down prior to generating typematic support with AIM_FKRate. A value of 0 indicates typematic keystroke support is disabled.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: The error return codes for this function are as follows:

8113H INVALID_PARAMETER
810CH GENERAL_FAILURE.

See the DosDevIOCtl error return codes in the *OS/2 2.0 Control Program Programming Reference* for more information.

Remarks: Session-switching and saving/restoring notifications are not given when switching between non-full screen sessions (that is, between Presentation Manager sessions and text-windowed sessions). Termination and creation notifications are given in all cases.

For requests with an Action parameter value in the range of 1 – 64, refer to the DosStartSession API in the *OS/2 2.0 Control Program Programming Reference* for Session ID/Type parameter values and descriptions. Any request received by a physical device driver that does not match the Length/Action parameter values, as defined above, is returned to the caller with an ERROR_INVALID_PARAMETER error code.

Function 60H – Query Monitor Support

This function queries the monitor support.

Parameter Packet Format

Field	Length
Command Information	BYTE

Command Information Reserved. Must be set to 0.

Data Packet Format

Field	Length
Reserved. Set to 0.	BYTE

Returns: None.

Remarks: This request is used to query a device driver for monitor support. The physical device driver returns the system error, MONITORS_NOT_SUPPORTED, if it does not support character monitors. If monitors are supported, then it returns NO_ERROR (00H).

Appendixes

Appendix A. Running OS/2 Version 1.3 16-Bit Physical Device Drivers on OS/2 2.0

The following items are possible problems that can occur when running OS/2 Version 1.3 16-bit physical device drivers on OS/2 2.0.

Use of Physical Addresses - PhysToUVirt

Physical device drivers which use the DevHlp service, PhysToUVirt, will notice changes in these areas:

- Returned offset (in BX) can no longer equal 0.
- The *OS/2 Version 1.3 I/O Subsystems and Device Support Volume 1 Device Drivers* (erroneously) stated that the result found in BX, on return from the call to PhysToUVirt, would always be 0.
- In OS/2 2.0, BX is not always equal to 0 on return from PhysToUVirt.
- Mapped areas that are > 60KB might not be fully mapped if their beginning offset is > 0. For example, a 64KB object that begins at n bytes offset from a 4KB page boundary, results in a segment/descriptor that has a length of $64KB - n$.

Direct Call to Physical Device Drivers

Physical device drivers that setup their own GDT call gate will not work in OS/2 2.0. The solution is to use the DevHlp, DynamicAPI, in order to have the system generate a GDT call gate into the physical device driver.

Direct Writing of GDT Selectors

Physical device drivers that setup GDT descriptors for private use will not work in OS/2 2.0. The solution is to use one of three Device Helper services, PhysToGDTSelector, PhysToUVirt, or PhysToVirt. Which service to use must be based on the context in which the memory is to be accessed.

Physical Device Drivers that Need Real Mode

Physical device drivers that need to switch to real mode will fail. OS/2 2.0 defines virtual device drivers that co-ordinate Multiple DOS Sessions access to physical devices with the physical device driver. Refer to the *OS/2 2.0 Virtual Device Driver Reference* for further information.

Physical Device Drivers Support of DOS Applications

OS/2 physical device drivers, which were written to support a single DOS application can fail in OS/2 2.0 if they start supporting to several DOS applications at the same time. This concerns physical device drivers that maintain addresses of items within a DOS program. With OS/2 2.0 Multiple DOS Sessions support, an address within a DOS program might not always refer to the same DOS Session.

If a physical device driver supports multiple DOS applications and needs to maintain data relative to each, the device driver must be modified to the VDD/PDD model. See "Inter-Device Driver Communication" on page 5-7.

Appendix B. Using Advanced BIOS

Physical device drivers that invoke Advanced BIOS (ABIOS) by using one of the following transfer conventions are commonly known as physical BIOS device drivers:

- BIOS Transfer Convention
- Operating System Transfer Convention.

For the BIOS Transfer Convention, the BIOS common entry points are invoked to locate the specific start, interrupt, or timeout entry points for the requesting Logical ID.

For the Operating System Transfer Convention, the specific start, interrupt, or timeout entry points must be located and called for the requesting Logical ID. OS/2 internal device drivers use the Operating System Transfer Convention to invoke BIOS services. User-written, installable device drivers can use either the BIOS Transfer Convention or the Operating System Transfer Convention. Device Helper services (DevHlps) are provided for both calling conventions.

For performance reasons, BIOS device drivers should call BIOS services with processor interrupts enabled.

Device Driver Data Segment

OS/2 2.0 recommends that an BIOS device driver use its data segment to contain the BIOS request blocks and data transfer buffers. The device driver data segment is located in memory. The operating system guarantees addressability to the data segment regardless of the processor mode (protect or real). The physical device driver assumes that the physical location of the device driver data segment will not move. This allows physical data transfers to take place to buffers within the data segment.

By using the data segment, the physical device driver can create logical addressability to these data areas (for BIOS) in a mode-independent manner, and without interrupt disable time considerations. Physical data transfers to buffers outside of the device driver's data segment can take place if the buffer is locked. As an alternate method, if the buffer is passed on a DosRead or DosWrite file system call, the physical address can be converted to a GDT Selector and used in place of the virtual address. The file system guarantees to lock down any buffer passed on a DosRead or DosWrite API. Therefore, this alternate method increases performance by removing the need to *double buffer* the data.

Obtaining a Logical ID

During its initialization, an BIOS device driver must obtain the Logical ID (LID) for its physical device. The allocation of a LID is managed by the operating system. This ensures that the physical device driver receives a unique LID for its device type. The operating system provides a DevHlp function, GetLIDEntry, to obtain the LID for a device driver. The DevHlp, GetLIDEntry, finds the LID for the specified Device ID and allocates it to the calling device driver.

An BIOS device driver maps its device name to a unit within a Logical ID (LID). It receives a DEINSTALL request for its device name. This implies a single unit of a LID. To honor the DEINSTALL request, it must relinquish the LID through use of the DevHlp, FreeLIDEntry, at deinstall time.

Note: To release a LID means to release all units under that LID. For a LID that has multiple units, the physical device driver must discontinue support of all units under the LID. If multiple units correspond with multiple device headers in the device driver data segment, the device driver must note which device header corresponds to each unit in the DEINSTALL LID and discontinue support.

The operating system prohibits access the LID for system services. Operating system management of LID access is similar to the management of hardware interrupt levels or I/O ports.

Calling Advanced BIOS Services

For BIOS device drivers that use the Operating System Transfer Convention, a DevHlp service, `ABIOSCall`, is provided to invoke BIOS with the mode-specific correct set of parameters. The physical device driver passes the BIOS request block pointer, its LID, and the BIOS primary function (start, interrupt, or timeout) to the DevHlp, `ABIOSCall`. This sets up the stack for the call to BIOS, and calls the BIOS function.

For BIOS device drivers that use the BIOS Transfer Convention, a DevHlp service, `ABIOSCommonEntry`, is provided to invoke the BIOS Common Entry Points. The physical device driver passes the mode-specific pointer to the BIOS request block, and the BIOS primary function (common start, common interrupt, or common timeout) to the DevHlp service. The DevHlp service sets up the stack, and calls the requested BIOS common entry point.

Note: The return code of the BIOS function is in the BIOS request block.

Mapping Device Names to LID

Having identified its LID and the number of devices or physical units the LID represents, the physical device driver must map each of its device names to a unit within the LID and support all units under the LID. All physical device drivers are known to the operating system by device names whether these names correspond to the ASCII string device name in the header for character device drivers, or to the logical units (which correspond to drive letters) in the header for block device drivers.

For character device drivers using BIOS, the device name represents a single device identified by the Logical ID (LID). When a character device driver has a single device driver header, its device name must be mapped to the first unit of the LID it obtained. If the character device driver has one device header but its LID had multiple units, the rest of the units are not used.

When a character device driver has multiple device driver headers, the operating system calls the strategy routine entry point for each header during device driver initialization:

1. The first entry point called must map its device name to the first unit of the LID.
2. The second entry point called must map its device name to the next unit of the LID.
3. If the LID that was obtained by the first entry point has only one unit, the second entry point must obtain another LID, and map its device name to the first unit of the second LID.
4. The physical device driver must start with the first LID, and consume all the units before going to the next LID.

For a block device driver, the number of units can be placed in the first byte. This is optional since the OS/2 operating system fills this field during device driver initialization. For block device drivers using BIOS, the number of units is equivalent to the number of devices (or units) under the LID. The block device driver must obtain the necessary number of LID/units for the number of logical units it supports, and map the first logical unit to the first-LID/first-unit, the second logical unit to the next available LID/unit, and so forth.

Handling ABIOS Requests

Refer to “Writing a Physical Device Driver Using Advanced BIOS” on page B-3, for information on handling requests to ABIOS. A physical device driver must assume that it owns all outstanding ABIOS request blocks for a given Logical ID. During interrupt-time processing, the physical device driver must call ABIOS for each outstanding request that is *incomplete-waiting-on-interrupt*. This is one of the reasons that a Logical ID is not shared among physical device drivers.

Writing a Physical Device Driver Using Advanced BIOS

To determine the basic structure of the physical device driver, certain design points must be identified:

- The kind of device to be supported (character or block).
- The nature of the I/O to the device; synchronous or staged, Program I/O (PIO), or Direct Memory Access (DMA). A staged request can be staged on a time delay, staged on an interrupt, or both. *Staged on a time delay* means the operation involves waiting for a specific length of time before the operation can be continued, or is completed. *Staged on an interrupt* means the operation involves waiting for an interrupt to occur.

PIO or DMA refers to the type of addressing required for data transfers. PIO is done using virtual addresses, which are also referred to as logical addresses, of the form:

segment/selector:offset.

DMA is performed using physical addresses, which are 32-bit numbers, indicating the data transfer location in memory.

- The maximum number of devices.
- The maximum number of interrupt levels.

These items determine the nature of the physical device driver, that is, how the task-time and interrupt-time portions of the device driver relate to each other, and which of the DevHlp services will be used for blocking, queueing, timers, and so forth. The type and number of devices generally indicate the logical device names (for example, COM1, LPT1) that the physical device driver supports:

- A device type is identified by its ABIOS-architected Device ID.
- A specific device is identified by a Logical ID (LID), and unit number under that LID.
- I/O to the device is performed by calling the ABIOS entry point (start, interrupt, or timeout) that corresponds to the particular LID.
- Parameters are passed to an ABIOS service through a request block structure.
- I/O requests can be synchronous (run to completion), or staged (run until blocked, waiting for an interrupt or time).
- Staged requests may have well-defined time delays between certain stages.
- Data transfers can use either physical addresses, or virtual addresses.

Before using ABIOS services and during initialization, a physical device driver must identify every LID for which it will accept requests. To do this, the physical device driver uses the architected ABIOS Device ID for its device.

The physical device driver uses the DevHlp, GetLIDEntry, to search through the ABIOS common data area looking for the LID that corresponds to the given Device ID. In general, by making repeated calls to GetLIDEntry, and counting the number of units supported by each LID it obtains, the physical device driver determines how many supported devices are configured in the system. The physical device driver only

processes interrupts and requests for its maximum number of supported devices. Any LID of the device driver's device type that is left over must be unclaimed so another physical device driver can support it.

A physical device driver knows which LID corresponds to a given logical device name (for example, COM1) because of the rule forcing the operating system logical device names to be in the same order as the LID entries for an associated Device ID. For example, assuming one unit per LID, an installable printer device driver supports LPT3 (the third printer) by locating the third LID that corresponds to Device ID of printer.

The physical device driver must determine which interrupt level each LID uses by using the BIOS function, Return LID Parameters. The device driver registers interrupt handler entry points for the interrupt levels that it supports with the DevHlp SetIRQ. It keeps a list of every LID that corresponds to each interrupt handler.

Note: If the physical device driver supports multiple devices, and the number of interrupt levels for those devices exceed the number of supported interrupt levels, the device driver ignores any LID that it cannot support since too many different interrupt levels are required.

At task time, when the device driver strategy entry point for a given device header receives a request packet, the device driver knows which logical device name and LID (and unit number) correspond to that entry point.

The device driver strategy routine sets up an BIOS request block, and uses the DevHlp BIOSCall to invoke the Advanced BIOS START routine to begin the requested BIOS function. BIOS requires that the return code field, in the BIOS request block, be initialized to FFFFH. BIOS sets the return code to its appropriate value.

Note: Either portion of the device driver, the task-time strategy routine or the interrupt handler, can start an BIOS request. For simplicity, the example uses the strategy routine as the caller of the Advanced BIOS START service.

Interrupt During START: If the request is staged on an interrupt, then BIOS sets the return code appropriately, only when the particular service is ready to be resumed through the Advanced BIOS interrupt routine. The device driver strategy routine must also set a flag to indicate whether a request has completed the Start request to the point at which the strategy routine interrogates the return code. This must be done to accommodate the case where the interrupt occurs after BIOS updates the return code, but before the device driver strategy routine interrogates the return code. In this case, the device driver interrupt handler is invoked by the interrupt, and can take appropriate action on the request block, even though the device driver strategy routine has not completed processing the request block.

For example, if the strategy routine is expected to block the request until the interrupt occurs, but the interrupt handler is invoked before the strategy routine is able to block, the interrupt handler needs to flag the fact that the interrupt handler has already processed the request block. The strategy routine, when it gets control, should not check the return code in the request block, and does not have to block, because the request is already completed. It then sets up the request packet with the return information, and returns the completed request to the kernel.

This example would be more complex, if when the strategy routine got control, the request was still incomplete (as would occur in a multi-staged request). The strategy routine would ignore the return code because the request block is already at a different stage than the start, but it might still have to block.

Interrupt After START: For a staged ABIOS request that must wait for the interrupt associated with the specified LID to occur after the request is started, the return code of the request is set to *stage-on-interrupt* by the ABIOS START function. This indicates that the request is incomplete. Several requests for this LID can start, and be waiting for the device interrupt. These incomplete requests are commonly referred to as outstanding requests for the LID.

Note: The request is considered to be an outstanding request for the LID, even if the Start service has not returned control to the caller of the Start service.

A physical device driver does not assume that the return codes for an ABIOS request occur in any given order. The return code should always be checked to determine what actions to perform on the request block.

When the device driver interrupt handler is invoked by the device interrupt, it knows which LID is associated with the interrupt level. The interrupt handler must individually examine each LID associated with the interrupt level. For a LID, the interrupt handler must process all outstanding staged-on interrupt request blocks. That is, the interrupt handler is required by ABIOS to call the Advanced BIOS interrupt routine for every outstanding staged-on interrupt request block to completely process one LID. This includes a START request block in which the return code has been changed from FFFH to stage-on interrupt, but the Start service has not yet returned control to its caller. If one of the request blocks for the LID caused the interrupt, after the interrupt handler has called ABIOS with all the outstanding request blocks owned by this LID, the interrupt handler does not need to process any other LID associated with this interrupt level.

Advanced BIOS requires that physical device drivers adhere to the following rules for sharing interrupt levels:

ABIOS REQUEST BLOCK RULE: The interrupt handler for a particular Logical ID (LID), when invoked by the operating system, must call Advanced BIOS for each ABIOS request block that is stage-on-interrupt, even if one of the request blocks gets the return indicator that the interrupt belongs to it.

ABIOS EOI PLACEMENT RULE: The EOI must be issued after all ABIOS staged-on interrupt request blocks have been processed for the LID that owns the interrupt.

ABIOS LID IRQ RULE: Advanced BIOS defines only one interrupt level for each LID. If a physical device driver handles more than one LID on the same interrupt level, the physical device driver could choose to register only one interrupt handler for any LID on that level. In this case, the operating system calls the handler only once when the interrupt occurs. The handler must manage the processing of more than one LID in order to determine if it owns the interrupt processing for them. In this case, the device driver interrupt handler should be aware of the Fairness Criteria problem. A LID at the end of the interrupt handler's list does not get as much service as a very active LID at the front of the list.

If a given LID has no outstanding ABIOS request blocks, the physical device driver calls the Advanced BIOS Default Interrupt service for that LID. The Default Interrupt service resets the interrupt condition for that LID if the LID falsely caused the interrupt. It then returns to the device driver interrupt handler, indicating that the interrupt belonged to the LID.

If there is at least one outstanding ABIOS request block for a given LID, ABIOS automatically invokes the Default Interrupt service if the LID generates a false interrupt. The physical device driver must be able to process the false interrupt return code for any call to the ABIOS interrupt routine. This return code indicates that the interrupt belonged to the LID, was reset by ABIOS, and the physical device driver is responsible for issuing the EOI, and returning to the operating system as owning the interrupt.

The device driver interrupt handler can issue the EOI (through the DevHlp EOI function) only after completely processing the LID that owns the interrupt, or after the LID's Default Interrupt service indicates that the LID's device caused the interrupt. The interrupt handler must process all outstanding requests under the LID that owns the interrupt, even after finding a request block which indicates that it caused the

interrupt. The interrupt handler can stop processing any LID for this interrupt only when the interrupt is claimed by a LID, either by a request under the owning LID, or by the owning LID's Default Interrupt service. Once the interrupt handler issues the EOI after completely processing a LID, another LID requesting service at the current interrupt level would create another interrupt. Once the EOI is issued, the interrupt handler can be re-entered at its entry point. If the interrupt handler is reentered, it must process every LID, including the one that is near completion or just completed.

In order to keep the pre-EOI processing time to a minimum, the interrupt handler should issue its EOI before it sets the return information in the operating system request packet, or before it begins processing a request packet queued by the device driver strategy routine.

If the interrupt handler did not find any LID that claimed the given interrupt with a request block, or by a Default Interrupt service, the interrupt handler must exit, indicating that the interrupt did not belong to it; that is, the interrupt was not caused by any LID that is owned by the physical device driver.

Eventually, the return code from BIOS will show that the BIOS request block is complete. The physical device driver can then clear the device driver request packet queue, take the next request packet, and try to start another BIOS request block.

The physical device driver supports the timeout requirements of BIOS with the DevHlp SetTimer, and the DevHlp TickCount. Instead of counting every clock tick, the device driver uses TickCount to force its timer tick entry point to receive control as infrequently as possible. The design of the BIOS TIMEOUT function is in one-second increments.

Spurious Interrupts

To handle a spurious interrupt in an edge-triggered interrupt environment, reset the interrupt at the 8259 interrupt controller (EOI), issue the global rearm if sharing interrupts, and enable processor interrupts. Generally, these actions would be taken by the last interrupt handler in the list of interrupt handlers.

To handle a spurious interrupt in a level-sensitive interrupt environment, reset the interrupt condition at the device, issue the EOI, and enable processor interrupts.

Advanced BIOS (ABIOS) provides the capability to reset a spurious interrupt at the device through the use of an Advanced BIOS default interrupt handler for the Logical ID (LID). The handler calls the BIOS default interrupt handler for its LID if there are no outstanding incomplete-waiting-on-interrupt request blocks. The BIOS default interrupt handler will indicate either that the interrupt condition was successfully reset or that the interrupt did not belong to the device referenced by the LID. If the BIOS handler replies that the device was successfully reset, the interrupt handler must issue the EOI and return as owning the interrupt.

Where there are incomplete-waiting-on-interrupt request blocks outstanding, BIOS keeps track of which interrupts are expected, and will automatically service a spurious interrupt when called by the interrupt handler. The handler must call BIOS with every request block that is incomplete-waiting-on-interrupt for a LID, even if the first one returns an indication that it performed some processing. The handler must also be able to process the spurious interrupt return code from any one of these calls to the BIOS interrupt service.

For a physical device driver that directly interfaces to the device, it must check the device for the interrupt condition, even if the interrupt condition does not correspond to an outstanding I/O request. If the device had caused the spurious interrupt, the interrupt handler must reset the interrupting condition at the device, issue the EOI, and return as owning the interrupt.

Note: If the device causing the spurious interrupt in the level-sensitive interrupt environment is not identified and reset, the interrupt level will interrupt continuously. This is a feature of the 8259 interrupt controller operating in level-sensitive mode. If no device driver that is registered for an interrupting interrupt level claims the interrupt, the OS/2 interrupt manager is forced to disable the

interrupt level at the 8259 interrupt controller since it cannot clear the interrupt at the device. If the interrupt is not disabled, the device will continue to generate spurious interrupts, the interrupt manager will continue to call all device drivers registered on the interrupt level, and all lower-priority interrupts will fail to be serviced.

Address Conversion Using DevHlp Services

To get a logical address to place in an ABIOS request block:

1. Call PhysToVirt with ES:DI for the converted address
2. Store the converted address in the ABIOS request block
3. Call the ABIOS service

For more information on interfaces to specific ABIOS functions, refer to the *Personal System/2 and Personal Computer BIOS Interface Technical Reference*.

Glossary

This glossary defines the terms used in this book. It includes terms and definitions from the *IBM Dictionary of Computing*, SC20-1699 as well as terms specific to the Presentation Manager but it is not a complete glossary for OS/2 Version 2.0.

A

ABIOS. Advanced BIOS. See *BIOS*.

action. One of a set of defined tasks a computer performs. Users request the application to perform an action in several ways: typing a command, pressing a function key or selecting the action name from an action bar or menu.

action point. The current position on the screen at which the pointer is pointing. (Contrast with *hotspot* and *input focus*.)

active program. A program currently running on the computer. See also *interactive program*, *noninteractive program* and *foreground program*.

active window. The window with which the user is currently interacting.

alphanumeric video output. Output to the logical video buffer when the video adapter is in text mode and the logical video buffer is addressed by an application as a rectangular array of character cells.

ANSI. American National Standards Institute.

APA. All points addressable.

API. Application programming interface. The formally defined programming language that is between an IBM application program and the user of the program. See also GPI.

ASCII. American National Standard Code for Information Interchange.

ASCIIZ. A string of ASCII characters that is terminated with a byte containing the value 0 (null).

aspect ratio. In computer graphics, the width-to-height ratio of an area, symbol or shape.

asynchronous. (1) Without regular time relationship. (2) Unexpected or unpredictable with respect to the execution of a program's instructions.

attributes. Characteristics or properties that can be controlled, usually to obtain a required appearance; for example, the color of a line. See also *segment attributes*.

AVIO. Advanced Video Input/Output.

B

background color. The color in which the background of a graphic primitive is drawn.

Bezier curves. A mathematical technique of specifying smooth continuous lines and surfaces which require a starting point and a finishing point with several intermediate points that influence or control the path of the linking curve. Named after Dr. P. Bezier.

BIOS. Microcode that controls basic hardware operations (for example, interactions with hard disk drives, diskette drives, and the keyboard).

bitmap. A representation in memory of the data displayed on an APA device, usually the screen.

border. A visual indication (for example, a separator line or a background color) of the boundaries of a window.

button. A mechanism on a *pointing device*, such as a mouse, used to request or initiate an action. Contrast with *pushbutton* and *radio button*.

C

CASE statement. Provides, in C/2, the body of a window procedure. There is one CASE statement for each message type written to take specific actions.

CGA. Color graphics adapter.

chained list. Synonym for *linked list*.

character. A letter, digit or other symbol.

character code. The means of addressing a character in a character set, sometimes called *code point*.

check box. A control window shaped like a square button on the screen, that can be in a checked or unchecked state. It is used to select one or more items from a list. Contrast with *radio button*.

check mark. The symbol (✓) that is used to indicate a selected item on a pull-down.

child window. A window that is positioned relative to another window (either a main window or another child window). Contrast with *parent window*.

choice. An option that can be selected. The choice can be presented as text, as a symbol (number or letter) or as an icon (a pictorial symbol).

class. See *window class*.

class style. The set of properties that apply to every window in a window class.

client area. The area in the center of a window that contains the main information of the window.

clipboard. An area of main storage that can hold data being passed from one Presentation Manager application to another. Various data formats can be stored.

clipping. In computer graphics, removing those parts of a display image that lie outside a given boundary.

clip limits. The area of the paper that can be reached by a printer or plotter.

code page. An assignment of graphic characters and control function meanings to all code points.

code point. Synonym for *character code*.

command. (1) The name and parameters associated with an action that can be performed by a program. A command is one form of action request. Users type in the command and enter it. (2) An action users request to interact with the command area.

command area. An area composed of two elements: a command field prompt and a command entry field.

command entry field. An entry field in which users type commands. The entry field is preceded by a command field prompt. These two elements make up the command area.

command line. On a display screen, a display line (usually at the bottom of the screen) in which only commands can be entered.

command prompt. A field prompt showing the location of the command entry field in a panel.

Common Programming Interface. A consistent set of specifications for languages, commands and calls to enable applications to be developed across all SAA environments. See also *Systems Application Architecture*.

Common User Access. A set of rules that define the way information is presented on the screen and the techniques for the user to interact with the information.

Compatibility Kernel. Portion of IBM Operating System/2 kernel that exists to support DOS INT 20, 21, 25, 26, 27 functionality; acts as interface to common kernel functionality, for example, file system.

Contention. State where a DOS Session attempts to access a serially reusable resource (for example, a

COM device) when another DOS Session or an IBM Operating System/2 application is using the resource.

Context Hook. Similar to force flags in IBM Operating System/2, these are events, signaled by a virtual device driver, that are processed at task time. Forcing an IRET and simulating an NMI can fall into this category.

control. The means by which an operator gives input to an application. A *choice* corresponds to a control.

Control Panel. In the Presentation Manager, a program used to set up user preferences that act globally across the system.

Control Program. The basic function of OS/2 including DOS emulation and the support for keyboard, mouse and VIO.

control window. A class of window used to handle a specific kind of user interaction. Radio buttons and check boxes are examples.

CPI. See *Common Programming Interface*.

cursor. A symbol displayed on the screen and associated with an input device. The cursor indicates where input from the device will be placed. Types of cursors include text cursors, graphics cursors and selection cursors. Contrast with *pointer* and *input focus*.

D

data structure. (ISO) The syntactic structure of symbolic expressions and their storage allocation characteristics.

DBCS. See *double-byte character set*.

default value. A value used when no value is explicitly specified by the user. For example, in the graphics programming interface, the default line type is 'solid'.

Desktop Manager. In the Presentation Manager, a window from which users can start one or more listed programs.

desktop window. The window corresponding to the physical device against which all other types of windows are established.

device context. A logical description of a data destination such as memory, metafile, display, printer or plotter. See also *direct device context*, *information device context*, *memory device context*, *metafile device context*, *queued device context* and *screen device context*.

device driver. A file that contains the code needed to use a device such as a display, printer or plotter.

Device Helpers (DevHlp). Kernel services (memory, hardware interrupt, software interrupt, queuing, semaphore,...) provided to physical device drivers.

dialog. The interchange of information between a computer and its user through a sequence of requests by the user and the presentation of responses by the computer.

dialog box. A type of window that contains one or more controls for the formatted display and entry of data. Also known as a *pop-up window*. A modal dialog box is used to implement a pop-up window.

Dialog Box Editor. A what-you-see-is-what-you-get (WYSIWYG) editor that creates dialog boxes for communicating with the application user.

dialog item. A component (for example, a menu or a button) of a dialog box. Dialog items are also used when creating dialog templates.

dialog template. The definition of a dialog box which contains details of its position, appearance and window ID; and the window ID of each of its child windows.

direct manipulation. The action of using the mouse to move objects around the screen. For example, moving files and directories about in the File Manager.

directory. A type of file containing the names and controlling information for other files or other directories.

DOS Session. A session created by the OS/2 Operating System that supports the independent execution of a DOS program. The DOS program appears to run independent of any other program in the system.

DOS Session breakpoint. A mechanism to regain control from a DOS Session; a V86 IRET frame is edited to point to an illegal V86 instruction, and the original CS:IP is saved in a table indexed by the address of the illegal instruction. A DOS Session breakpoint is a byte of memory visible in each DOS Session containing an illegal instruction.

double-byte character set (DBCS). A set of characters in which each character is represented by two bytes. Languages such as Japanese, Chinese and Korean which contain more characters than can be represented by 256 code points require double-byte character sets. As each character requires two bytes, the entering, displaying and printing of DBCS characters requires hardware and software that can support DBCS.

dragging. In computer graphics, moving an object on the display screen as if it were attached to the pointer.

drop. To fix the position of an object that is being dragged by releasing the select button of the pointing device.

DTL. See *dialog tag language*.

E

EBCDIC. Extended binary-coded decimal interchange code.

EGA. Extended graphics adapter.

entry field. A panel element in which users type information. Compare to *selection field*.

entry panel. A defined panel type containing one or more entry fields and protected information such as headings, prompts and explanatory text.

Event semaphore. A signaling mechanism, which enables a thread or process to notify waiting threads or processes that an event has occurred.

extended help. A facility that provides users with information about an entire application panel rather than a particular item on the panel.

entry-field control. The means by which the application receives data entered by the user in an entry field. When it has the input focus, it displays a flashing pointer at the position where the next typed character will go.

exit. The action that terminates the current function and returns the user to a higher level function. Repeated exit requests return the user to the point from which all functions provided to the system are accessible. Contrast with *cancel*.

extended-choice selection. A mode that allows the user to select more than one item from a window. Not all windows allow extended choice selection. Contrast with *multiple-choice selection*.

F

field-level help. Information specific to the field on which the cursor is positioned. This help function is "contextual" because it provides information about a specific item as it is currently used. The information is dependent upon the context within the work session.

File Manager. In the Presentation Manager, a program that displays directories and files and allows various actions on them.

file specification. The full identifier for a file which includes its file name, extension, path and drive.

font. A particular size and style of typeface that contains definitions of character sets, marker sets and pattern sets.

foreground program. The program with which the user is currently interacting. Also known as *interactive program*.

frame. The part of a window that can contain several different visual elements specified by the application but drawn and controlled by the Presentation Manager. The frame encloses the client area.

frame styles. Different standard window layouts provided by the Presentation Manager.

full-screen application. An application program that occupies the whole screen.

function key. A key that causes a specified sequence of operations to be performed when it is pressed; for example, F1 and Alt-K.

G

Global Descriptor Table (GDT). Defines code and data segments available to all tasks in an application.

glyph. A graphic symbol whose appearance conveys information.

GPI. Graphics Programming Interface. The formally defined programming language that is between an IBM graphics program and the user of the program. See also *API*.

graphics. A picture defined in terms of graphic primitives and graphics attributes.

graying. The indication that a choice on a pull-down is unavailable.

group. A collection of logically-connected controls. For example, the buttons controlling paper size for a printer. See also *program group*.

H

handle. An identifier that represents an object, such as a device or window.

hardcopy. Physical output (such as paper, slides or transparencies) from a device.

hard error. An error condition that requires either that the system be reconfigured or that the source of the error be removed before the system can resume reliable operation.

heap. An area of free storage available for dynamic allocation by the program. Its size varies depending on the storage requirements of the program.

help. A function that provides information about a specific field, an application panel or information about the help facility. It provides field help when the cursor is on a selection or entry field in an application panel or another help panel. It provides information about the application panel, called *extended help*, when the cursor is not in an interactive field.

help index. A facility that allows the user to select topics for which help is available.

help panel. A panel with information to assist users that is displayed in response to a help request from the user.

help window. A Common User Access defined secondary window that displays information when the user requests help.

hit testing. The means of identifying which window is associated with which input device event.

hook. A mechanism by which procedures are called when certain events occur in the system. For example, the filtering of mouse and keyboard input before it is received by an application program.

hook chain. A sequence of hook procedures that are "chained" together so that each event is passed in turn to each procedure in the chain.

hotspot. The part of the pointer that must touch an object before it can be selected. This is usually the tip of the pointer. Contrast with *action point*.

I

icon. A pictorial representation of an item the user can select. Icons can represent items (such as a document file) that the user wants to work on and actions that the user wants to perform. In the Presentation Manager, icons are used for data objects, system actions and minimized programs.

Icon Editor. The Presentation Manager-provided tool for creating icons.

information device context. A logical description of a data destination other than the screen (for example, a printer or plotter) but where no output will occur. Its purpose is to satisfy queries. See also *device context*.

input focus. The area of the screen that will receive input from an input device (typically the keyboard).

input router. OS/2 internal process that removes messages from the system queue.

interactive graphics. Graphics that can be moved or manipulated by a user at a terminal.

Interactive program. A program that is running (active) and is ready to receive (or is receiving) input from the user. Also known as a *foreground program*. Compare with *active program* and contrast with *noninteractive program*.

Interrupt Request Flag. A bit in the 8259 PIC controller that indicates an interrupt is pending on particular level. The VPIC also maintains a virtual *interrupt request* flag for each interrupt level for each DOS Session.

Interrupt Service Flag. A bit in the 8259 PIC controller that indicates an interrupt request is being serviced. It is cleared when the PIC is sent EOI. The VPIC maintains a virtual *interrupt service* flag indicating that a simulated interrupt is in-progress in a DOS Session.

IOPL. Input/Output Privilege Level. Allows part of a Ring 3 application or device driver to execute at Ring 0.

IOPM. Input/Output Permission Map. Bitmap pointed to by the 386 TSS that controls, on a per-port basis, whether an IN/OUT instruction causes a fault.

IRQ. A term that broadly means an *interrupt request level*. This term refers to either pending or in-service interrupt requests, or to a specific level (for example, IRQ 4).

J

journal. A special-purpose file that is used to record changes made in the system.

K

kanji. A graphic character set used in Japanese ideographic alphabets.

kerning. The design of graphics characters so that their character boxes overlap. Used to space text proportionally.

keys help. A facility that gives users a listing of all the key assignments for the current application.

L

language support procedure. Function provided by the Presentation Interface for applications that do not or cannot (as in the case of COBOL and FORTRAN programs) provide their own dialog or window procedures.

LIFO Stack. A data structure from which data is retrieved in last-in, first-out order.

linked list. A list in which the data elements may be dispersed but in which each data element contains

information for locating the next. Synonym for *chained list*.

list box. A control window containing a vertical list of selectable descriptions.

list panel. A defined panel type that displays a list of items from which users can select one or more choices and then specify one or more actions to work on those choices.

Local Descriptor Table (LDT). Defines code and data segments specific to a single task.

LVB. Logical Video Buffer.

M

main window. The window that is positioned relative to the desktop window.

maximize. A window-sizing action that makes the window the largest size possible.

media window. The part of the physical device (display, printer or plotter) on which a picture is presented.

memory device context. A logical description of a data destination that is a memory bitmap. See also *device context*.

menu. A type of panel that consists of one or more selection fields. Also called a *menu panel*.

message. 1. In Presentation Manager, a packet of data used for communication between the Presentation Interface and windowed applications.

2. In a user interface, information not requested by users but presented to users by the computer in response to a user action or internal process.

Messages on status, problems or user actions from a computer application should be distinguished from a "message" or note sent to users by other users over a communications link.

message filter. The means of selecting which messages from a specific window will be handled by the application.

message queue. A sequenced collection of messages to be read by the application.

metafile. The generic name for the definition of the contents of a picture. Metafiles are used to allow pictures to be used by other applications.

metafile device context. A logical description of a data destination that is a metafile which is used for graphics interchange. See also *device context*.

metalanguage. A language used to specify another language.

mickey. A unit of measurement for physical mouse motion whose value depends on the mouse device driver currently loaded.

minimize. A window-sizing action that makes the window the smallest size possible. In the Presentation Manager, minimized windows are represented by icons.

mixed character string. A string containing a mixture of one-byte and *kanji* or *hangeul* (two-byte) characters.

mnemonic. A method of selecting an item on a pull-down by means of typing the highlighted letter in the menu item.

modal dialog box. The type of control that allows the operator to perform input operations on only the current dialog box or one of its child windows. Also known as a *serial dialog box*. Contrast with *parallel dialog box*.

modeless dialog box. The type of control that allows the operator to perform input operations on any of the application's windows. Also known as a *parallel dialog box*. Contrast with *modal dialog box*.

mouse. A hand-held device that is moved around to position the pointer on the screen.

Multiple DOS Session. A system service that coordinates the concurrent operation of multiple DOS Sessions. See also *DOS Session*.

Multiple Virtual DOS Machine. Deprecated term for multiple DOS Sessions.

multiple-choice selection. A mode that allows users to select any number of choices including none at all. See also *check box*. Contrast with *extended-choice selection*.

multi-tasking. The concurrent processing of applications or parts of applications. A running application and its data are protected from other concurrently running applications.

N

named pipe. A named object that provides client-to-server, server-to-client or duplex communication between unrelated processes. Contrast with *unnamed pipe*.

noninteractive program. A program that is running (active) but is not ready to receive input from the user. Compare with *active program* and contrast with *interactive program*.

null-terminated string. A string of (n + 1) characters where the (n + 1)th character is the 'null' character (X'00') and is used to represent an n-character string with implicit length. Also known as 'zero-terminated' string and 'ASCIIZ' string.

O

object window. A window that does not have a parent but which may have child windows. An object window cannot be presented on a device.

open. To start working with a file, directory or other object.

output area. The area of the output device within which the picture is to be displayed, printed or plotted.

owner window. A window into which specific events that occur in another (owned) window are reported.

P

paint. The action of drawing or redrawing the contents of a window.

panel. A particular arrangement of information grouped together for presentation to the user in a window.

panel area. An area within a panel that contains related information. The three major panel areas defined by the Common User Access are the action bar, the function key area and the panel body.

panel body. The portion of a panel not occupied by the action bar, function key area, title or scroll bars. The panel body may contain protected information, selection fields and entry fields. The layout and content of the panel body determine the panel type.

panel body area. The part of a window not occupied by the action bar or function key area. The panel body area may contain information, selection fields and entry fields. Also known as *client area*.

panel body area separator. A line or color boundary that provides users with a visual distinction between two adjacent areas of a panel.

panel definition. A description of the contents and characteristics of a panel. Thus, a panel definition is the application developer's mechanism for predefining the format to be presented to users in a window.

panel ID. A panel element located in the upper left-hand corner of a panel body that identifies that particular panel within the application.

panel title. A panel element that identifies the information in the panel.

paper size. The size of paper, defined in either standard U.S. or European names (for example, A, B, A4) and measured in inches or millimeters respectively.

parallel dialog box. See *modeless dialog box*.

parent window. The window relative to which one or more child windows are positioned. Contrast with *child window*.

pel. The smallest area of a display screen capable of being addressed and switched between visible and invisible states. Synonymous with *display point*, *pixel* and *picture element*.

physical device driver (PDD). Device driver responsible for primary control of a physical device. This corresponds very closely to the OS/2 1.00 dual-mode device driver, except that it does not support the DOS environment directly. Rather, it supports interfaces that can be used by virtual device drivers to support &dosapps..

pick. To select part of a displayed object using the pointer.

pipe. See *named pipe*, *unnamed pipe*.

pixel. Synonym for *pel*. See *pel*

plotter. An output device that uses pens to draw its output on paper or transparency foils.

pointer. The symbol displayed on the screen that is moved by a pointing device such as a *mouse*. The pointer is used to point at items that users can select. Contrast with *cursor*.

pointing device. A device (such as a mouse) used to move a pointer on the screen.

pointings. Pairs of x-y coordinates produced by an operator defining positions on a screen with a pointing device such as a *mouse*.

polyfillet. A curve based on a sequence of lines. It is tangential to the end points of the first and last lines and tangential also to the midpoints of all other lines. See also *fillet*.

polyline. A sequence of adjoining lines.

pop. To retrieve an item from a last-in-first-out stack of items. Contrast with *push*.

pop-up window. A window that appears on top of another window in a dialog. Each pop-up window must be completed before returning to the underlying window.

Presentation Manager. The OS/2 Control Program plus the visual component that presents, in windows, a graphics-based interface to applications and files installed and running in OS/2.

primary window. The window in which the main dialog between users and the application takes place. In a multi-programming environment, each application starts in its own primary window. The primary window remains for the duration of the application although the panel displayed will change as the user's dialog moves forward. See also *secondary window*.

print job. The result of sending a document or picture to be printed.

Print Manager. In the Presentation Manager, the part of the spooler that manages the spooling process. It also allows users to view print queues and to manipulate print jobs.

process. An instance of an executing application and the resources it is using.

program details. Information about a program that is specified in the Desktop Manager window and is used when the program is started.

program group. In the Presentation Manager, several programs that can be acted upon as a single entity.

program name. The full file specification of a program. Contrast with *program title*.

program title. The name of a program as it is listed in the Desktop Manager window. Contrast with *program name*.

pull-down. An extension of the *action bar* that displays a list of choices available for a selected action bar choice. After users select an action bar choice, the pull-down appears with the list of choices. Additional *pop-up windows* may appear from pull-down choices to further extend the actions available to users.

push. To add an item to a last-in-first-out stack of items. Contrast with *pop*.

pushbutton. A control window shaped like a rounded-corner rectangle and containing text, that invokes an immediate action such as 'enter' or 'cancel'.

Q

queue. A list of print jobs waiting to be printed.

queued device context. A logical description of a data destination (for example, a printer or plotter) where the output is to go through the spooler. See also *device context*.

R

radio button. A control window shaped like a round button on the screen that can be in a checked or unchecked state. It is used to select a single item from list. Contrast with *check box*.

reentrant. The attribute of a program or routine that allows the same copy of the program or routine to be used concurrently by two or more tasks.

reference phrase. A word or phrase that is emphasized in a device-dependent manner in order to inform the user that additional information for the word or phrase is available.

reference phrase help. Provides help information on a selectable phrase.

refresh. To update a window with changed information to its current status.

resource. The means of providing extra information used in the definition of a window. A resource can contain definitions of fonts, templates, accelerators and mnemonics; the definitions are held in a resource file.

restore. To return a window to its original size or position following a sizing or moving action.

reverse video. A form of alphanumeric highlighting for a character, field or cursor in which its color is exchanged with that of its background. For example, changing a red character on a black background to a black character on a red background.

RGB. Red-green-blue. For example "RGB display".

roman. Relating to a type style with upright characters.

S

screen. The physical surface of a workstation or terminal upon which information is presented to users.

screen device context. A logical description of a data destination that is a particular window on the screen. See also *device context*.

scroll bar. A control window, horizontally or vertically aligned, that allows the user to scroll additional data into an associated panel area.

scrollable entry field. An entry field larger than the visible field.

scrollable selection field. A selection field that contains more choices than are visible.

scrolling. Moving a display image vertically or horizontally in a manner such that new data appears at

one edge as existing data disappears at the opposite edge.

secondary window. A type of window associated with the primary window in a dialog. A secondary window begins a secondary and parallel dialog that runs at the same time as the primary dialog.

segment. See *graphics segment*.

select. To mark or choose an item. Note that *select* means to mark or type in a choice on the screen; *enter* means to send all selected choices to the computer for processing.

select button. The button on a pointing device, such as a mouse, that is pressed to select a menu choice. Also known as button 1.

selection cursor. A type of cursor used to indicate the choice or entry field users want to interact with. It is represented by highlighting the item it is currently positioned on.

selection field. A field containing a list of choices from which the user can select one or more.

semaphore. An object used by multi-threaded applications for signaling purposes and for controlling access to serially reusable resources.

separator. See *panel body area separator*.

serial dialog box. See *modal dialog box*.

serially reusable resource (SRR). A logical resource or object that can be accessed by only one task at a time.

session. A routing mechanism for user interaction via the console.

shadow box. The area on the screen that follows mouse movements and shows what shape the window will take if the mouse button is released.

shutdown. In the Desktop Manager, the procedure required before the computer is switched off to ensure that data is not lost.

sibling windows. Child windows that have the same parent window.

slider box. An area on the scroll bar that indicates the size and position of the visible information in a panel area in relation to the information available. Also known as *thumb mark*.

spline. A sequence of one or more Bezier curves.

spooler. A program that intercepts the data going to printer devices and writes it to disk. The data is printed or plotted when it is complete and the required device

is available. The spooler prevents output from different sources being intermixed.

standard window. A collection of windows that form a panel.

static control. The means by which the application presents descriptive information (for example, headings and descriptors) to the user. The user cannot change this information.

suballocation. The allocation of a part of one extent for occupancy by elements of a component other than the one occupying the remainder of the extent.

switch. An action that moves the input focus from one area to another. This can be within the same window or from one window to another.

switch list. See Task List.

symbolic identifier. A text string that equates to an integer value in an include file that is used to identify a programming object.

System Menu. In the Presentation Manager, the pull-down in the top left corner of a window that allows it to be moved and sized with the keyboard.

system queue. This is the master queue for all pointer device or keyboard events.

Systems Application Architecture. A formal set of rules that enables applications to be run without modification in different computer environments.

T

tag. A markup language word and its attributes that are entered in the source file to identify parts of a panel or other dialog objects.

Task List. In the Presentation Manager, the list of programs that are active. The list can be used to switch to a program and to stop programs.

template. An ASCII-text definition of an action bar and pull-down menu held in a resource file or as a data structure in program memory.

text. Characters or symbols.

text cursor. A symbol displayed in an entry field that indicates where typed input will appear.

text window. Also known as the VIO window. The environment in which OS/2 runs AVIO applications.

thread. A unit of execution within a process.

thumb mark. The portion of the scroll bar that describes the range and properties of the data that is currently visible in a window. Also known as a *slider box*.

tilde. A mark used to denote the character that is to be used as a mnemonic when selecting text items within a menu.

title bar. The area at the top of a window that contains the window title. The title bar is highlighted when that window has the input focus. Contrast with *panel title*.

Tree. In the Presentation Manager, the window in the File Manager that shows the organization of drives and directories.

U

unnamed pipe. A circular buffer created in memory; used by related processes to communicate with one another. Contrast with *named pipe*.

User Shell. A component of OS/2 that uses a graphics-based, windowed interface to allow the user to manage applications and files installed and running under OS/2.

V

VGA. Video graphics array.

VIO. Video Input/Output.

virtual device driver (VDD). Maintains shadow state of hardware, if necessary. Allows a DOS Session to execute *in a window or in the background* by intercepting direct device access and simulating that device.

Virtual DevHlp (VDH). Kernel (linear memory, paging, hardware interrupt, event control, port control) services provided to virtual device drivers.

Virtual DOS Machine. Depreciated term for DOS Session. See *DOS Session*.

virtual memory (VM). Addressable space that is apparent to the user as the processor storage space but not having a fixed physical location.

Virtual Programmable Interrupt Controller (VPIC). Virtualizes the 8259 Programmable Interrupt Controller (PIC). A special virtual device driver, in that it provides services to other virtual device drivers.

visible region. A window's presentation space clipped to the boundary of the window and the boundaries of any overlying window.

V86 Mode. Virtual 8086 mode of the 80386.

W

wild-card character. The global file-name characters ? or *.

window. A rectangular area of the screen through which a panel or portion of a panel is displayed. A window can be smaller than or equal in size to the screen. Windows can overlap on the screen and give the appearance of one window being on top of another.

window class. The grouping of windows whose processing needs conform to the services provided by one window procedure.

window coordinates. The means by which a window position or size is defined; measured in device units or *pels*.

window procedure. Code that is activated in response to a message.

window rectangle. The means by which the size and position of a window is described in relation to the desktop window.

window style. The set of properties that influence how events related to a particular window will be processed.

workstation. A display screen together with attachments such as a keyboard, a local copy device or a tablet.

WYSIWYG. What You See Is What You Get. A capability that enables text to be displayed on a screen in the same way that it will be formatted on a printer.

Z

z-order. The order in which sibling windows are presented. The topmost sibling window obscures any portion of the siblings that it overlaps; the same effect occurs down through the order of lower sibling windows.

zooming. In graphics applications, the process of increasing or decreasing the size of picture.

Index

A

- ABIOS EOI placement rule B-5
- ABIOS LID IRQ rule B-5
- ABIOS request block rule B-5
- ABIOSCall, DevHlp 17-12
- ABIOSCommonEntry, DevHlp 17-13
- access authorization 8-27
- additional function support 8-2
- additional port support 8-2
- advanced BIOS services
 - ABIOSCall, DevHlp 17-12
 - ABIOSCommonEntry, DevHlp 17-13
 - FreeLIDEntry, DevHlp 17-31
 - GetLIDEntry, DevHlp 17-39
- Advanced BIOS/device driver notes B-3
- AllocateCtxHook, DevHlp 17-14
- AllocCtxHook (AllocateCtxHook), DevHlp 17-14
- AllocGDTSelector, DevHlp 17-15
- AllocPhys, DevHlp 17-16
- AllocReqPacket, DevHlp 17-17
- ANSI and Code pages 14-58
- ANSI.SYS 14-54
- application I/O 2-1
 - See also character device monitors
 - See also IOPL
- ArmCtxHook, DevHlp 17-18
- ASYNC Control IOCTL Commands (Category 1) 18-7
- ASYNC IOCTL commands (Category 1)
 - Extended Query Bit Rate, 63H 18-41
 - Extended Set Bit Rate, 43H 18-11
 - Query COM Error, 6DH 18-48
 - Query COM Event Information, 72H 18-49
 - Query COM Status, 64H 18-42
 - Query Current Bit Rate, 61H 18-39
 - Query DCB Parameters, 73H 18-50
 - Query Enhanced Mode Parameters, 74H 18-53
 - Query Line Characteristics, 62H 18-40
 - Query Modem Control Output Signals, 66H 18-44
 - Query Modem Input Signals, 67H 18-45
 - Query Number of Chars in Receive Queue, 68H 18-46
 - Query Number of Chars in Transmit Queue, 69H 18-47
 - Query Transmit Data Status, 65H 18-43
 - Set Bit Rate, 41H 18-8
 - Set Break OFF, 45H 18-15
 - Set Break ON, 4BH 18-20
 - Set DCB Parameters, 53H 18-21
 - Set Enhanced Mode Parameters, 54H 18-36
 - Set Line Characteristics, 42H 18-9
 - Set Modem Control Signals, 46H 18-16
 - Start Transmit, 48H 18-19
 - Stop Transmit, 47H 18-18

ASYNC IOCTL commands (Category 1) (continued)

- Transmit Byte Immediate, 44H 18-13
- ASYNC Notes 18-23
- asynchronous communications/COM 8-1
- asynchronous communications/IOCTL summary 18-7
- AT bus adapter support 8-2
- AttachDD, DevHlp 17-19
- attachment support 8-2
- attribute field 3-3
- attribute field bits
 - clock device 3-4
 - device type 3-3
 - format 3-3
 - generic IOCTL request 3-4
 - Get/Set Logical Device 3-4
 - IDC bit 3-3
 - NULL 3-4
 - removable media 3-4
 - shared 3-3
 - standard input 3-4
 - standard output 3-4
- automatic control 8-5
- automatic monitoring 8-5
- automatic receive flow control 8-6
- automatic receive flow control/XON-XOFF 8-6
- automatic transmit flow control 8-6
- automatic transmit flow control/XON-XOFF 8-6
- AUX 8-22, 8-29

B

- Beep, DevHlp 17-20
- binary - ASCII 8-24
- binary data 8-24
- BIOS Parameter Block (BPB) 3-3, 15-5
- bit rate/return 18-18
- block device drivers 2-1
- block device unit code field 15-1
- Block, DevHlp 17-21
- boot sector format 15-9
- BPB 10-7
- BPB, BIOS Parameter Block 3-3, 15-5
- break 8-7
- break replacement character/processing 8-7
- break signal/off 18-14
- BUILD BPB, strategy command 15-9
- busy bit 15-2

C

- call Advanced BIOS services, device drivers B-2
- Capabilities Bit Strip, device header 3-4
- category codes, generic IOCTL 18-1

- Category 1 IOCtl Commands 18-7
- Category 10 IOCtl Commands 18-168
- Category 11 IOCtl Commands 18-171
- Category 3 IOCtl Commands 18-55, 18-57, 18-59
- Category 4 IOCtl Commands 18-65
- Category 5 IOCtl Commands 18-105
- Category 5, Function 4DH 18-110
- Category 5, Function 4EH 18-111
- Category 5, Function 6EH 18-117
- Category 7 IOCtl Commands 18-118
- Category 8 IOCtl Commands 18-144
- Category 9 IOCtl Commands 18-161
- character device data stream 6-2
- character device drivers 2-1
- Character Device Monitor Control IOCtl Commands (Category 10) 18-168
- character device monitors
 - limitations 6-2
 - monitor buffer sizes 6-14
 - monitor data, format 6-15
 - monitor dispatcher 6-4
 - monitor dispatcher device helper 6-25
 - monitor functions 6-9
 - monitor mechanism 6-4
 - monitor problems and solutions 6-22
 - monitor processes 6-5
 - monitor support in device drivers 6-25
 - monitor termination 6-18
 - monitoring data streams 6-2
 - performance considerations 6-17
 - position in a monitor chain 6-15
 - printer monitor 6-24
 - thread priorities 6-16
 - well-behaved monitor applications 6-19
- character monitor performance 13-6
- character queue management 5-3
 - QueueFlush, DevHlp 17-65
 - QueueInit, DevHlp 17-66
 - QueueRead, DevHlp 17-67
 - QueueWrite, DevHlp 17-68
- clear to send (CTS) 8-2
- clock device bit 3-4
- CLOCKS device driver
 - CLOCK\$ device time format 9-1
- CLOSE 8-25
- CLOSE DEVICE, strategy command 15-15
- CloseEventSem, DevHlp 17-23
- CLOSE/automatic receive flow control 8-25
- CLOSE/break processing 8-25
- CLOSE/DTR 8-25
- CLOSE/interrupt level 8-25
- CLOSE/last level close 8-25
- CLOSE/RTS 8-25
- CLOSE/transmit hardware 8-25
- CLOSE/transmit immediate processing 8-25
- Close_Mouse, mouse IDC request 12-4
- code page number 18-100

- code page support 8-24
- code page support, parallel port device driver 13-3
- COM 8-1
- command codes
 - command code field 15-1
 - device driver 15-4
 - summary 15-1, 15-4
- command-specific field, request header 15-3
- commands, IOCtl 18-3
- communications device driver 8-1
- COM/break signal/on 18-19
- COM/IOCtl summary 18-7
- CONFIG.SYS
- Console device drivers
 - device drivers, screen and keyboard 11-1
 - keyboard initialization 11-2
 - keyboard run time operation 11-3
 - keystroke monitor data packet 11-3
 - keystroke monitors 11-3
 - physical keyboard device driver (KBD\$) 11-1
- contexts, physical device driver 3-1
- control screen cursor 14-54
- control sequences 14-54
- controlling display mode 14-57
- CTS 8-2
- CTS/return 18-44
- cursor 14-54
- cursor backward 14-56
- cursor control 14-55
- cursor control sequences
 - cursor backward 14-56
 - cursor down 14-55
 - cursor forward 14-55
 - cursor position 14-55
 - cursor position report 14-56
 - cursor up 14-55
- device status report 14-56
- erase in display 14-57
- erase in line 14-57
- horizontal position 14-56
- keyboard key reassignment 14-58
- restore cursor position 14-57
- save cursor position 14-56
- set graphics rendition 14-57
- vertical position 14-56
- cursor up 14-55
- custom device support 8-2

D

- data carrier detect (DCD) 8-2
- data set ready (DSR) 8-2
- data stream monitoring 6-2
- data terminal ready (DTR) 8-2
- data translation 8-24
- DCD 8-2
- DCD/return 18-44

- DDINSTALL 7-1
 - See *a/so* external device drivers
- DDP 7-1
 - See *a/so* external device drivers
- DEINSTALL hardware interrupts 15-20
- DEINSTALL Logical IDs B-1
- DEINSTALL, strategy command 15-20
- DeRegister, DevHlp 6-28, 17-24
 - See *a/so* character device monitors
- desktop window
 - definition of X-2
- DevDone, DevHlp 17-25
- DevEnable 14-5
- DevHlp character queue structure 5-3
- DevHlp services
 - See Device Helper (DevHlp) services
- DevHlp SetIRQ 4-6
- DevHlps
 - See Device Helper (DevHlp) services
- device attribute 3-3
- device buffer control IOctls 18-171
- device data streams 6-2
- device driver
 - application access 2-1
 - application I/O 2-1
 - attribute 3-3
 - building 3-6
 - command codes 15-4
 - commands 5-6
 - components 3-5
 - converting DOS Session addresses within IOctls requests 5-6
 - design issues, physical device drivers 5-1
 - device header 3-2
 - DosDevIOctl 2-1
 - hardware interrupt handler 3-5
 - header 3-2
 - interrupt handler 5-4
 - IOctl commands 5-6
 - memory management 5-2
 - monitor support 6-25
 - monitor support, character device driver 5-4
 - OS/2 requests 5-3
 - performance considerations, physical device drivers 5-1
 - RAM semaphores 5-2
 - request packet 3-5, 15-1
 - request packet queue management 5-1
 - requests to OS/2 5-3
 - routines 3-5
 - semaphore management 5-2
 - status word 15-2
 - step-by-step (building) 3-6
 - strategy routine 3-5
 - system semaphores 5-2
 - timer handler 3-6
 - writing 3-6
- device driver architecture
 - components of a physical device driver 3-5
- device driver data segment, ABIOS B-1
- device driver examples 17-58
- device driver file image 3-2
- device driver functions
 - DosBeep 4-2
 - DosCaseMap 4-2
 - DosChgFilePtr 4-2
 - DosClose 4-2
 - DosDelete 4-2
 - DosDevConfig 4-2
 - DosDevIOctl 4-2
 - DosFindClose 4-2
 - DosFindFirst 4-2
 - DosFindNext 4-2
 - DosGetEnv 4-2
 - DosGetInfoSeg 4-2
 - DosGetMessage 4-2
 - DosOpen 4-2
 - DosPutMessage 4-2
 - DosQCurDir 4-2
 - DosQCurDisk 4-2
 - DosQFileInfo 4-2
 - DosQFileMode 4-2
 - DosRead 4-2
 - DosSMRegisterDD 4-2
 - DosWrite 4-2
- device driver header fields
 - attribute 3-3
 - capabilities bit strip 3-4
 - header 3-3
 - name/unit 3-4
 - next device header 3-3
 - strategy routine 3-4
- device driver interrupt sharing 4-4
- device driver monitor buffers 6-29
 - See *a/so* character device monitors
- device driver monitor chain buffer 6-26
 - See *a/so* character device monitors
- device driver monitor code requirements 6-30
 - See *a/so* character device monitors
- device driver notes/using ABIOS B-3
- device driver notification routine 6-26
 - See *a/so* character device monitors
- device driver profile (DDP) 7-1
 - See *a/so* external device drivers
- device driver program model 3-2
- device drivers
 - disk device driver 10-1
- device drivers, base video handlers 14-1
- device drivers, video 14-1
 - chaining 14-1
 - identification 14-1
 - running system installation 14-1
 - write support 14-1
- device driver/close consideration 8-8

- device driver/defaults 8-8
- device driver/Mode Utility 8-8
- device driver/open consideration 8-8
- device driver/states 8-8
- device driver, block 2-1
- device driver, character 2-1
- device driver, DOS Session 2-3
- device driver, EGA.SYS 14-39
- device driver, interrupt sharing rules 4-5
- device driver, multiple block devices 2-1
- device driver, multiple character devices 2-1
- device driver, previous-level 4-3
- device driver, queueing requests 5-1
- device field, next 3-3
- device header 3-2
- device helper services 17-1
- Device Helper (DevHlp) services
 - ABIOSCall 17-12
 - ABIOSCommonEntry 17-13
 - AllocateCtxHook 17-14
 - AllocGDTSelector 17-15
 - AllocPhys 17-16
 - AllocReqPacket 17-17
 - ArmCtxHook 17-18
 - AttachDD 17-19
 - Beep 17-20
 - Block 17-21
 - CloseEventSem 17-23
 - DeRegister 17-24
 - DevDone 17-25
 - DynamicAPI 17-27
 - EOI 17-28
 - FreeCtxHook 17-29
 - FreeGDTSelector 17-30
 - FreeLIDEntry 17-31
 - FreePhys 17-32
 - FreeReqPacket 17-33
 - GetDescInfo 17-34
 - GetDeviceBlock 17-35
 - GetDOSVar 17-36
 - GetLIDEntry 17-39
 - InternalError 17-40
 - LinToGDTSelector 17-41
 - LinToPageList 17-42
 - Lock 17-43
 - MonFlush 17-45
 - MonitorCreate 17-46
 - MonWrite 17-48
 - OpenEventSem 17-49
 - PageListToGDTSelector 17-50
 - PageListToLin 17-52
 - PhysToGDTSel 17-54
 - PhysToGDTSelector 17-55
 - PhysToUVirt 17-56
 - PhysToVirt 17-57
 - PostEventSem 17-60
 - ProtToReal 17-61
 - PullParticular 17-62

Device Helper (DevHlp) services (*continued*)

- PullReqPacket 17-63
- PushReqPacket 17-64
- QueueFlush 17-65
- QueueInit 17-66
- QueueRead 17-67
- QueueWrite 17-68
- RealToProt 17-69
- Register 17-70
- RegisterBeep 17-71
- RegisterPDD 17-72
- RegisterStackUsage 17-73
- RegisterTmrDD 17-74
- ResetEventSem 17-75
- ResetTimer 17-76
- ROMCriticalSection 17-77
- Run 17-78
- SAVE_MESSAGE 17-26
- SchedClockAddr 17-79
- SemClear 17-80
- SemHandle 17-81
- SemRequest 17-83
- SendEvent 17-84
- SetIRQ 17-85
- SetROMVector 17-86
- SetTimer 17-87
- SortReqPacket 17-88
- TCYield 17-89
- TickCount 17-90
- Unlock 17-91
- UnPhysToVirt 17-92
- UnSetIRQ 17-93
- VerifyAccess 17-94
- VideoPause 17-95
- VirtToLin 17-96
- VirtToPhys 17-97
- VMAIloc 17-98
- VMFree 17-100
- VMGlobalToProcess 17-101
- VMLock 17-103
- VMProcessToGlobal 17-105
- VMSetMem 17-107
- VMUnlock 17-108
- Yield 17-109

Device Monitor IOCTL commands (Category 10)

- Register A Monitor, 40H 18-168

device monitor limitations 6-2

device monitors

- See character device monitors

device names 8-22

- device names/Advanced BIOS 8-22
- device names/COMn 8-22
- device names/determination 8-22
- device names/IBM PS/2 8-22
- device names/Logical ID 8-22
- device names/none 8-22
- device names/40: 8-22

- device status report 14-56
- device support diskette 7-1
 - See *also* external device drivers
- device type bit 3-3
- device-dependent device driver 12-6
- devices, ill-behaved 4-5
- devices, well-behaved 4-5
- DEVICE = parameter string
 - device driver examples 17-58
 - driver 15-1
 - physical device driver header 3-2
 - physical device driver program model 3-2
 - replacing character device 4-3
- Disable_Device mouse IDC request 12-7
- Disable_Support mouse IDC request 12-3
- disk control IOCTLs 18-161
- disk control IOCTL, query device parameters 18-159
- disk control IOCTL, query physical device parameters 18-166
- disk (physical) control IOCTLs 18-161
- diskette control IOCTL, query device parameters 18-159
- diskette control IOCTL, query physical device parameters 18-166
- diskette device driver 10-1
- disk, (logical) control IOCTLs 18-144
- display, primary 14-3
- done bit 15-2
- DOS extended volume architecture
 - BPB and get device parameters 10-7
 - creating block devices 10-4
 - deleting block devices 10-4
 - extended DOS partition architecture 10-2
 - installing block devices 10-3
- DOS Session device driver 2-3
- DOS Session/considerations 8-29
- DOS Session/CTTY 8-29
- DOS Session/FAPI 8-29
- DOS Session/file system 8-29
- DOS Session/IOCTLs 8-29
- DOS Session/old device drivers 8-29
- DOS Session/printing 8-29
- DosDevIOCTL 2-1
- DosMonClose 6-13
 - See *also* character device monitors
- DosMonOpen 6-9
 - See *also* character device monitors
- DosMonRead 6-11
 - See *also* character device monitors
- DosMonReg 6-10, 11-3
 - See *also* character device monitors
- DosMonWrite 6-12
 - See *also* character device monitors
- DSR 8-2
- DSR/return 18-44
- DTR 8-2
- DTR and RTS/return 18-43

- DTR/CLOSE 8-6
- DTR/disable 8-6
- DTR/enable 8-6
- DTR/input handshaking 8-6
- DTR/OPEN 8-6
- dynamic link calls, device driver 4-2
- DynamicAPI, DevHlp 17-27

E

- EGA.SYS device driver 14-39
- Enable_Device mouse IDC request 12-7
- EOI rule 4-6
- EOI, DevHlp 17-28
- erase control sequences
 - erase in display 14-57
 - erase in line 14-57
- erase in display control sequence 14-57
- erase in line control sequence 14-57
- erasing control sequences 14-57
- error 8-7
- error bit 15-2
- error codes, status WORD 15-2
- error replacement character/processing 8-7
- error/return 18-47
- event notification 8-7
- event WORD/clear 18-48
- event WORD/return 18-48
- events
- examples of device drivers 17-58
- Extended DOS Partition 10-2
- extended help
 - definition of X-3
- extended partition 10-2
- extended screen and keyboard, using
 - control sequence syntax 14-54
 - cursor control sequences 14-55
 - limitations/restrictions 14-54
- extended startup record 10-2
- extended volume 10-2
- external device drivers
 - DDINSTALL.EXE 7-1
 - device driver profile (DDP) 7-1
 - device support diskette 7-1
 - installation 7-1

F

- field name 3-4
- field, attribute 3-3
- file system 8-25
- first level open 8-5
- fixed disk device driver 10-1
- flow control/automatic 8-6
- flow control/IOCTLs 8-6
- flow control/manual 8-6
- FLUSH 8-5

- format bit 3-3
- Free Physical Buffer/111H (273) 14-28
- FreeCtxHook, DevHlp 17-29
- FreeGDTSelector, DevHlp 17-30
- FreeLIDEntry, DevHlp 17-31
- FreePhys, DevHlp 17-32
- FreeReqPacket, DevHlp 17-33
- function codes
 - character queue management 5-3
 - memory management 5-2
 - semaphore management 5-2
 - services and device contexts 17-6
- function codes, generic IOCTL 18-1

G

- General Device Control IOCTL Commands (Category 11) 18-171
- General Device IOCTL commands (Category 11)
 - Flush Input Buffer, 01H 18-171
 - Flush Output Buffer, 02H 18-172
 - Query Monitor Support, 60H 18-177
 - System Notifications for Physical Device Drivers, 41H 18-173
- generic IOCTL commands
 - See ASYNC IOCTL commands (Category 1)
 - See General Device IOCTL commands (Category 11)
 - See Keyboard IOCTL commands (Category 4)
 - See Logical Disk IOCTL commands (Category 8)
 - See Mouse IOCTL commands (Category 7)
 - See Physical Disk IOCTL commands (Category 9)
 - See Pointer Draw IOCTL commands (Category 3)
 - See Printer IOCTL commands (Category 5)
 - See Screen IOCTL commands (Category 3)
 - See Video IOCTL commands (Category 3)
- generic IOCTLs/COM 18-7
- generic IOCTL, Category 8 10-7
- generic IOCTL, category 9 10-8
- GENERIC IOCTL, strategy command 15-17
- Get Device Parameters 10-7
- GET DRIVER CAPABILITIES, strategy command 15-24
- GET FIXED DISK/LOGICAL UNIT MAP, strategy command 15-22
- GET LOGICAL DRIVE MAP, strategy command 15-19
- GetDescInfo, DevHlp 17-34
- GetDeviceBlock, DevHlp 17-35
- GetDOSVar, DevHlp 17-36
- GetLIDEntry, DevHlp 17-39

H

- handlers, base video 14-1
- hardware id, keyboard 18-103
- hardware interrupt handler 3-5
- hardware interrupt interface, parallel port, printer 13-8
- hardware interrupt management 4-4

- hardware interrupt sharing 4-4
- hardware interrupts, DEINSTALL 15-20
- header, device driver 3-3
- horizontal position 14-56
- hot key 18-81, 18-97

I

- IDC entry point 3-4
- idcbt.IDC bit 3-3
- IDC, inter-device driver communication 5-7
- identification; video device handler 14-1
- ill-behaved device rule 4-5
- ill-behaved devices 4-5
- INIT mode 3-1
- InitEnable 14-7
- initialization 8-22
- initialization/CONFIG.SYS ordering 8-22
- initialization/DEINSTALL 8-22
- initialization/device names 8-22
- initialization/DEVICE = 8-22
- initialization/filenames 8-22
- initialization/IBM AT bus 8-24
- initialization/IBM PS/2 8-24
- initialization/interrupt level 8-23
- initialization/Logical ID 8-24
- initialization/multiple device drivers 8-22
- initialization/Personal Computer AT 8-23
- initialization/ROM BIOS 40: data area 8-24
- initialization/40: area 8-23
- initialization, PDD 14-38
- Initialize Environment/101H (257) 14-11
- INIT, strategy command 15-5
- INPUT FLUSH, strategy command 15-14
- input modem control signals 8-2, 8-6
- input sensitivity/DSR 8-6
- INPUT STATUS, strategy command 15-13
- INT return rule 4-5
- INT 14H 8-29
- INT 2FH, screen switch notification 14-54
- INT 21H interface, parallel port 13-7
- INT 21H interface, printer 13-7
- INT 33H - DOS Mode Mouse API
- inter-device driver communication 3-4
 - AttachDD, DevHlp 17-19
 - IDC bit 3-3
- InternalError, DevHlp 17-40
- interrupt driven 8-4
- interrupt handler
 - hardware 3-5
 - maximum per interrupt level 4-6
- interrupt handler, physical device driver 5-4
- interrupt management
 - EOI, DevHlp 17-28
 - SetIRQ, DevHlp 17-85
 - SetROMVector, DevHlp 17-86
 - UnSetIRQ, DevHlp 17-93

- Interrupt mode 3-1
- interrupt processing 4-4
- interrupt processing, 8259 enabling 4-5
- interrupt routine 4-1
- interrupt sharing 4-4
- interrupt sharing, device dependency 4-4
- interrupt sharing, getting an IRQ 4-5
- interrupt sharing, ill-behaved device 4-5
- interrupt sharing, processing interrupts 4-5
- interrupt sharing, restrictions 4-5
- interrupt sharing, rules 4-5
- interrupt sharing, well-behaved device 4-5
- interrupt sharing, 8259 enabling 4-5
- interrupt-time 3-1
- interrupts, nested 5-4
- interrupts, spurious B-6
- IOctl 8-4
 - Category 5, Function 4DH 18-110
 - Category 5, Function 4EH 18-111
 - Category 5, Function 6EH 18-117
 - character device monitor commands 18-168
 - command summary 18-3
 - commands 18-3
 - general device control 18-171
 - Keyboard control commands 18-65
 - Logical Disk Control 18-144
 - Mouse control commands 18-118
 - Parallel Port control commands 18-105
 - Physical Disk control commands 18-161
 - Pointer Draw control commands 18-55
 - Screen Control Commands 18-59
 - summary 18-3
 - Video control commands 18-57
- IOctl command summary 18-3
- IOctl commands, Keyboard Control 18-65
- IOctl commands, mouse control 18-118
- IOctl commands, parallel port control 18-105
- IOctl command, Pointer Draw Control 18-55
- IOctl command, Screen Control 18-59
- IOctl command, Video Control 18-57
- IOctl READ 15-11
- IOctl support of code page and font switching, printer
 - activate font 13-5
 - query active font 13-5
 - verify font 13-5
- IOctl WRITE 15-11
- IOctls/COM 18-7
- IOctl, character device monitor 18-168
- IOctl, disk control 18-161
- IOctl, general device control 18-171
- IOctl, logical disk control 18-144
- IOctl, monitor 18-168
- IOctl, physical disk control 18-161
- IOPL
- IRQ enforcement rule 4-5
- IRQ mask rule 4-5
- IRQ ownership rule 4-5

I/O services

- i/o support for the DOS Session 2-3
- i/o support for the OS/2 mode 2-2

J

- job title packet 13-14

K

- KBD 14-58
- KbdStringIn 14-58
- Kernel mode 3-1
- Keyboard Control IOctl Commands (Category 4) 18-65
- keyboard id 18-99
- Keyboard IOctl commands (Category 4)
 - Alter Keyboard LEDs, 5AH 18-86
 - Create a Logical Keyboard, 5DH 18-88
 - Destroy a Logical Keyboard, 5EH 18-89
 - Peek Character Data Record, 75H 18-95
 - Query Code Page Number, 78H 18-100
 - Query Input Mode, 71H 18-90
 - Query Interim Character Flags, 72H 18-91
 - Query Keyboard Code Page Info, 7AH 18-104
 - Query Keyboard Hardware ID, 7AH 18-103
 - Query Keyboard Type, 77H 18-99
 - Query Session Manager Hot Key, 76H 18-97
 - Query Shift State, 73H 18-92
 - Read Character Data Records, 74H 18-93
 - Set Code Page Number, 58H 18-84
 - Set Code Page, 50H 18-66
 - Set Input Mode, 51H 18-77
 - Set Interim Character Flags, 52H 18-78
 - Set KCB, 57H 18-83
 - Set NLS and Custom Code Page, 5CH 18-87
 - Set Read Notification, 59H 18-85
 - Set Session Manager Hot Key, 56H 18-81
 - Set Shift State, 53H 18-79
 - Set Typematic Rate and Delay, 54H 18-80
 - Translate Scan Code To ASCII, 79H 18-101
- keyboard reassignment 14-58
- keyboard run time operation 11-3
- keystroke monitors 11-3
- keys, reassign 14-58

L

- last level close 8-5
- LDT, local descriptor table 4-1
- length of request packet field 15-1
- line characteristics 8-7
- line characteristics/bit rate 8-7
- line characteristics/data bits 8-7, 18-9
- line characteristics/parity 8-7, 18-9
- line characteristics/return 18-39
- line characteristics/set 18-9
- line characteristics/stop bits 8-7, 18-9

- linkage field, request header 15-3
- LinToGDTSelector, DevHlp 17-41
- LinToPageList, DevHlp 17-42
- Lock, DevHlp 17-43
- logical block device 10-2
- Logical Disk Control IOCTL Commands (Category 8) 18-144
- Logical Disk IOCTL commands (Category 8)
 - Begin Format, 04H 18-149
 - Block Removable, 20H 18-150
 - Format and Verify track, 45H 18-156
 - Lock Drive, 00H 18-145
 - Query Device Parameters, 63H 18-159
 - Query Logical Map, 21H 18-151
 - Redetermine Media, 02H 18-147
 - Set Device Parameters, 43H 18-152
 - Set Logical Map, 03H 18-148
 - Unlock Drive, 01H 18-146
 - Write/Read/Verify Track, 44H, 64H, 65H 18-154
- Logical IDs, DEINSTALL B-1

M

- manual control 8-5
- manual monitoring 8-5
- marking 8-2
- master startup record, partition table 10-6
- MEDIA CHECK, strategy command 15-7
- memory addressability 5-2
- memory management 5-2
 - AllocGDTSelector, DevHlp 17-15
 - AllocPhys, DevHlp 17-16
 - FreePhys, DevHlp 17-32
 - Lock, DevHlp 17-43
 - PhysToGDTSelector, DevHlp 17-55
 - PhysToGDTSel, DevHlp 17-54
 - PhysToUVirt, DevHlp 17-56
 - PhysToVirt, DevHlp 17-57
 - Unlock, DevHlp 17-91
 - UnPhysToVirt, DevHlp 17-92
 - VerifyAccess, DevHlp 17-94
 - VirtToPhys, DevHlp 17-97
- mode 14-57
- mode of operation
 - reset mode (RM) 14-58
 - set graphics rendition (SGR) 14-57
 - set mode (SM) 14-57
- mode of operation control sequences
 - set graphics rendition 14-57
- Mode Utility/performance 8-29
- modem control input signals/return 18-44
- modem control output signals/return 18-43
- modem control signals 18-15
- modes, physical device driver 3-1
- MonFlush, DevHlp 6-28, 17-45
 - See *also* character device monitors
- monitor applications, well-behaved 6-19
- monitor buffers, sizes 6-14
- monitor data structures
 - keystroke 11-3
- monitor data, format 6-15
- monitor dispatcher 6-4
- monitor dispatcher device helper 6-25
 - See *also* character device monitors
 - DeRegister 6-28
 - MonFlush 6-28
 - MonitorCreate 6-25
 - MonWrite 6-27
 - Register 6-26
- monitor dispatcher interface 13-14
- monitor functions 6-9
 - See *also* character device monitors
 - DosMonClose 6-13
 - DosMonOpen 6-9
 - DosMonRead 6-11
 - DosMonReg 6-10
 - DosMonWrite 6-12
- monitor IOCTLs 18-168
- monitor management
 - DeRegister, DevHlp 17-24
 - MonFlush, DevHlp 17-45
 - MonitorCreate, DevHlp 17-46
 - MonWrite, DevHlp 17-48
 - Register, DevHlp 17-70
- monitor mechanism 6-4
- monitor problems and solutions 6-22
- monitor processes 6-5
- monitor support 8-24
- monitor support in character device drivers 5-4, 6-6, 6-25
- monitor support, limitations 6-2
- monitor termination 6-18
- monitor thread priorities 6-16
- monitor-job title packet 13-14
- MonitorCreate, DevHlp 6-25, 17-46
 - See *also* character device monitors
- monitoring data streams 6-2
- monitors
 - See character device monitors
- monitors, keystroke 11-3
- MonWrite, DevHlp 6-27, 17-48
 - See *also* character device monitors
- Mouse Control IOCTL Commands (Category 7) 18-118
- mouse device driver
 - devices supported 12-1
 - overview 12-1
- Mouse devices 12-1
- Mouse IOCTL commands (Category 7)
 - Assign New Mouse Event Mask, 54H 18-123
 - Display Mode Change, 51H 18-119
 - Mark Collision Area, 58H 18-127
 - Notification of Mode Switch Completion, 5DH 18-132
 - Query Event Queue Status, 64H 18-137
 - Query Mouse Device Driver Status Flags, 62H 18-135

Mouse IOCTL commands (Category 7) (continued)

- Query Mouse Device's Motion Sensitivity, 61H 18-134
- Query Mouse Event Mask, 65H 18-138
- Query Mouse Scaling Factors, 66H 18-139
- Query Number of Buttons Supported, 60H 18-133
- Query Physical Mouse Device Driver Level/Version, 6AH 18-142
- Query Pointer Screen Position, 67H 18-140
- Query Pointer Shape, 68H 18-141
- Query Pointing Device ID, 6BH 18-143
- Read Mouse Event Queue, 63H 18-136
- Reassign Mouse Scaling Factors, 53H 18-122
- Set OS/2 Mode Address of Pointer Draw Device Driver, 5AH 18-129
- Set Physical Mouse Device Driver Status Flags, 5CH 18-131
- Set Pointer Shape, 56H 18-124
- Specify/Replace Pointer Position, 59H 18-128
- Unmark Collision Area, 57H 18-126
- mouse monitors
 - device 6-2
 - keystroke 11-3
 - print 13-14
- Mouse physical device driver 12-1
- mouse pointer draw implementation
- MOUSE\$ 12-1
- multiple character device support per device driver 3-3
- multiple device headers per driver 2-1, 3-3
- multiple requests 8-4
- multiple requests/ordering 8-5

N

- name/units field 3-4
- nested interrupts 5-4
 - RegisterStackUsage, DevHlp 17-73
- NONDESTRUCTIVE READ NO WAIT, strategy command 15-12
- notes, ASYNC 18-23
- NULL bit 3-4

O

- obtaining a Logical ID, device drivers B-1
- OFF 8-2
- ON 8-2
- OPEN 8-25
- OPEN DEVICE, strategy command 15-15
- OpenEventSem, DevHlp 17-49
- OPEN/first level open 8-25
- OPEN/initialize port 8-25
- OPEN/interrupt level 8-25
- OPEN/last level close 8-25
- OPEN/timer tick 8-25
- Open_Mouse, mouse IDC request 12-4

- OS/2 device driver operations 4-1
- OS/2 functions, see functions also
 - device driver 4-2
- OUTPUT FLUSH, strategy command 15-14
- output handshaking/CTS 8-6
- output handshaking/DCD 8-6
- output handshaking/DSR 8-6
- output modem control signals 8-2
- output modem control signals/automatic control 8-6
- output modem control signals/manual control 8-6
- OUTPUT STATUS, strategy command 15-13
- overlapped input output 8-5

P

- PageListToGDTSelector, DevHlp 17-50
- PageListToLin, DevHlp 17-52
- Parallel Port Control IOCTLs Commands (Category 5) 18-105
- parallel port IRQ timeout value 18-117
- Parallel Port (Printer) device driver
 - code page support 13-3
 - hardware interrupt interface 13-8
 - INT 21H interface 13-7
 - IOCTL support of code page and font switching 13-4
 - monitor dispatcher notification interface 13-9
 - Parallel Port IOCTL functions 13-8
 - Parallel Port IRQ Timeout Processing 13-6
 - parallel port (printer) monitors 13-3
 - position of parallel port monitors 13-4
 - replacing the parallel port device driver 13-2
 - request packet interface 13-7
 - reserved device names 13-2
- partition table 10-6
- PARTITIONABLE FIXED DISKS, strategy command 15-21
- PDD initialization 14-38
- performance 8-29
- performance/configuration 8-29
- performance/hardware overrun 8-29
- performance/receive queue overrun 8-29
- physical device driver components 3-5
- physical device driver contexts 3-1
- physical device driver header 3-2
- physical device driver initialization 4-2
- Physical Disk Control IOCTL Commands (Category 9) 18-161
- Physical Disk IOCTL commands (Category 9)
 - Lock Physical Drive, 01H 18-162
 - Query Physical Device Parameters, 63H 18-166
 - Unlock Physical Drive, 01H 18-163
 - Write/Read/Verify Track, 44H, 64H, 65H 18-164
- physical interface 8-4
- physical Keyboard device driver 11-1
- PhysToGDTSelector, DevHlp 17-55
- PhysToGDTSel, DevHlp 17-54
- PhysToUVirt, DevHlp 17-56

- PhysToVirt rule 4-6
- PhysToVirt, DevHlp 17-27
- Pointer Draw Control IOCTL Commands (Category 3) 18-55
- Pointer Draw IOCTL commands (Category 3)
 - Query Pointer Draw Address, 72H 18-56
- pointing device support 12-1
- position rule 4-6
- PostEventSem, DevHlp 17-60
- previous-level device drivers 4-3
- previous-level device drivers, INIT 4-4
- previous-level device drivers, install 4-4
- previous-level device drivers, rules 4-3
- primary display, identification 14-3
- Print Screen/115H (277) 14-34
- print-monitor-job title packet 13-14
- Printer Activate Font IOCTL 18-109
- Printer IOCTL commands (Category 5)
 - Activate Font, 48H 18-109
 - Initialize Parallel Port, 46H 18-108
 - Query Active Font, 69H 18-115
 - Query Frame Control, 62H 18-112
 - Query Infinite Retry, 64H 18-113
 - Query Parallel Port Status, 66H 18-114
 - Set Frame Control, 42H 18-106
 - Set Infinite Retry, 44H 18-107
 - Verify Font, 6AH 18-116
- printer monitor 6-24
- printer query active font, Parallel Port IOCTL 18-115
- Printer Verify Font, Parallel Port IOCTL 18-116
- PRINTMONBUFSIZE= 13-6
- process management
 - Block, DevHlp 17-21
 - DevDone, DevHlp 17-25
 - Run, DevHlp 17-78
 - SAVE_MESSAGE, DevHlp 17-26
 - TCYield, DevHlp 17-89
 - Yield, DevHlp 17-109
- processing interrupts 4-4
- processing interrupts, interrupt sharing 4-5
- processor mode services
 - ProtToReal, DevHlp 17-61
 - RealToProt, DevHlp 17-69
- Process_Absolute mouse IDC request 12-3
- Process_Packet mouse IDC request 12-2
- ProtToReal, DevHlp 17-61
- PS/2/adaptor 8-1
- PS/2/Advanced BIOS 8-1
- PS/2/interrupt levels 8-1
- PS/2/number of ports 8-1
- PullParticular, DevHlp 17-62
- PullReqPacket, DevHlp 17-63
- PushReqPacket, DevHlp 17-64

Q

- query active font IOCTL, Parallel Port IOCTL 18-115

- Query Color Lookup Table/106H (262) 14-17
- Query Config Info/104H (260) 14-15
- Query Cursor Info/108H (264) 14-19
- Query DBCS Display Info/105H (262) 14-16
- Query Device Parameters 18-159
- Query Font/10AH (266) 14-21
- Query LVB Info/117H (279) 14-36
- query media sense, 60H 18-158
- Query Mode/10CH (268) 14-23
- Query Palette Registers/10EH (270) 14-25
- query parallel port IRQ timeout value IOCTL 18-117
- Query Physical Buffer/110H (272) 14-27
- Query Variable Info/112H (274) 14-29
- Query_Activity, mouse IDC request 12-5
- Query_Config mouse IDC request 12-6
- queue linkage field, request header 15-3
- QueueFlush, DevHlp 17-65
- queueing request packets, device driver 5-1
- QueueInit, DevHlp 17-66
- QueueRead, DevHlp 17-67
- QueueWrite, DevHlp 17-68

R

- RAM semaphores 5-2
- READ 8-4, 8-26
- READ (input), strategy command 15-11
- READ/multiple requests 8-26
- READ/ordered requests 8-26
- READ/queueing of requests 8-26
- READ/receive queue 8-26
- READ/receive queue buffer overrun 8-26
- READ/timeout processing 8-26
- Read_Disable mouse IDC request 12-7
- Read_Enable mouse IDC request 12-6
- RealToProt, DevHlp 17-69
- reassign keys 14-58
- receive queue 8-4
- receive queue/return size 18-45
- receive queue/return # characters 18-45
- RegisterBeep, DevHlp 17-71
- RegisterPDD, DevHlp 17-72
- RegisterStackUsage, DevHlp 17-73
- RegisterTmrDD, DevHlp 17-74
- Register, DevHlp 6-26, 17-70
 - See also character device monitors
- removable media bit 3-4
- REMOVABLE MEDIA, strategy command 15-16
- replacing character device drivers 4-3
- request handling 4-1
- request header 15-1
 - command-specific data field 15-3
 - queue linkage field 15-3
 - status word 15-1
- request header command-specific field 15-3
- request header queue linkage field 15-3
- request header status error codes 15-2

- request header Status field 15-1
- request packet 15-1
- request packet format 15-1
- request packet header 15-1
- request packet queue management 5-1
- request queue management
 - AllocReqPacket, DevHlp 17-17
 - FreeReqPacket, DevHlp 17-33
 - PullParticular, DevHlp 17-62
 - PullReqPacket, DevHlp 17-63
 - PushReqPacket, DevHlp 17-64
 - SortReqPacket, DevHlp 17-88
- request to send (RTS). 8-2
- RESET MEDIA, strategy command 15-18
- ResetEventSem, DevHlp 17-75
- ResetTimer, DevHlp 17-76
- restore cursor position 14-57
- Restore Environment/103H (259) 14-14
- RI 8-2
- RI/return 18-44
- RLSD 8-2
- RLSD - DCD 18-7
- ROM BIOS 40: data area 8-29
- ROMCritSection, DevHlp 17-77
- routines, strategy 3-4
- RS232-C 8-2, 18-7
- RS232-C port/IOctl summary 18-7
- RS232-C/COM 8-1
- RTS 8-2
- RTS/CLOSE 8-6
- RTS/disable 8-6
- RTS/enable 8-6
- RTS/input handshaking 8-6
- RTS/OPEN 8-6
- RTS/toggling on transmit 8-6
- rule, ABIOs EOI placement B-5
- rule, ABIOs LID IRQ B-5
- rule, ABIOs request block B-5
- rule, EOI 4-6
- rule, ill-behaved device 4-5
- rule, INT return 4-5
- rule, IRQ enforcement 4-5
- rule, IRQ mask 4-5
- rule, IRQ ownership 4-5
- rule, PhysToVirt 4-6
- rule, position 4-6
- rule, search rule 4-6
- rule, set IRQ 4-5
- rule, STI entry 4-5
- rule, system timer 4-5
- running version 1.3 16-bit PPDs on OS/2 2.0 A-1
- Run, DevHlp 17-78

S

- sample character device monitor 6-33
- See also character device monitors

- save cursor position 14-56
- Save Environment/102H (258) 14-12
- SAVE_MESSAGE, DevHlp 17-26
- SchedClockAddr, DevHlp 17-79
- schedule requests 8-4
- Screen Control IOctl Commands (Category 3) 18-59
- screen cursor control 14-55
- Screen IOctl commands (Category 3)
 - ABIOs Pass-Through, 74H 18-62
 - Allocate a Selector with Background Validation, 76H 18-64
 - Allocate a Selector with Offset, 75H 18-63
 - Allocate a Selector, 70H 18-60
 - Deallocate a Selector, 71H 18-61
- screen switch notification, DOS Session 14-54
- search rule 4-6
- semaphore management 5-2
 - SemClear, DevHlp 17-80
 - SemHandle, DevHlp 17-81
 - SemRequest, DevHlp 17-83
- SemClear, DevHlp 17-80
- SemHandle, DevHlp 17-81
- SemRequest, DevHlp 17-83
- SendEvent, DevHlp 17-84
- serial communications 18-7
- serial communications/COM 8-1
- serial pointing device considerations 12-2
- serial port/IOctl summary 18-7
- Set Color Lookup Table/107H (263) 14-18
- Set Cursor Info/109H (265) 14-20
- Set Font/10BH (267) 14-22
- set graphics rendition, control sequence 14-57
- set IRQ rule 4-5
- SET LOGICAL DRIVE MAP, strategy command 15-19
- Set Mode/10DH (269) 14-24
- Set Palette Registers/10FH (271) 14-26
- set parallel port timeout value IOctl 18-111
- set print job title IOctl 18-110
- Set Variable Info/113H (275) 14-31
- SetIRQ, DevHlp 17-85
- SetROMVector, DevHlp 17-86
- SetTimer, DevHlp 17-87
- shared bit 3-3
- shared interrupt architecture 4-4
- SHUTDOWN, strategy command 15-23
- SortReqPacket, DevHlp 17-88
- spacing 8-2
- spooler 13-4
- spooler support/COM to COM 8-24
- spooler support/COM to LPT 8-24
- spooler support/LPT to COM 8-24
- spurious interrupts B-6
- standard input bit 3-4
- standard output bit 3-4
- state of the COM port 8-7
- states/automatic receive flow control (XON-XOFF) 8-8
- states/automatic transmit flow control (XON-XOFF) 8-9

- states/bit rate 8-9
- states/break replacement character 8-9
- states/break replacement character processing 8-9
- states/COM error WORD 8-10
- states/data bits 8-10
- states/DTR 8-10
- states/DTR Control Mode 8-10
- states/DTR Disable 8-10
- states/DTR Enable 8-10
- states/DTR Input Handshaking 8-10
- states/error replacement character 8-11
- states/error replacement character processing 8-11
- states/event WORD 8-10
- states/input sensitivity using DSR 8-12
- states/null stripping 8-12
- states/output handshaking using CTS 8-12
- states/output handshaking using DCD 8-12
- states/output handshaking using DSR 8-12
- states/parity 8-13
- states/read timeout state 8-13
- states/read timeout value 8-14
- states/RTS 8-10
- states/RTS Control Mode 8-13
- states/RTS Disable 8-13
- states/RTS Enable 8-13
- states/RTS Input Handshaking 8-13
- states/RTS toggling on transmit 8-13
- states/stop bits 8-14
- states/transmit immediate 8-14
- states/transmitting break 8-14
- states/write timeout state 8-14
- states/write timeout value 8-15
- states/XOFF character 8-15
- states/XON character 8-15
- status field 15-1
- status field error codes 15-2
- status field, request header 15-1
- status WORD 15-1
- status WORD bits
 - busy 15-2
 - done 15-2
 - error 15-2
 - error code 15-2
- status/return 18-40
- STI entry rule 4-5
- strategy commands
 - BUILD BPB 15-9
 - DEINSTALL 15-20
 - FLUSH INPUT OR OUTPUT 15-14
 - GENERIC IOCTL 15-17
 - GET DRIVER CAPABILITIES 15-24
 - GET FIXED DISK/LOGICAL UNIT MAP 15-22
 - INIT 15-5
 - INPUT OR OUTPUT STATUS 15-13
 - LOGICAL DRIVE MAP 15-19
 - MEDIA CHECK 15-7
 - NONDESTRUCTIVE READ NO WAIT 15-12
 - OPEN OR CLOSE DEVICE 15-15

- strategy commands (*continued*)
 - PARTITIONABLE FIXED DISKS 15-21
 - READ 15-11
 - REMOVABLE MEDIA 15-16
 - RESET MEDIA 15-18
 - SHUTDOWN 15-23
 - WRITE 15-11
 - WRITE WITH VERIFY 15-11
- strategy routine 3-5, 4-1
- strategy routines 3-4
- support of previous-level device drivers 4-3
- system clock management
 - SchedClockAddr, DevHlp 17-79
- system performance/degradation 8-29
- system semaphores 5-2
- system services
 - GetDOSVar, DevHlp 17-36
 - ROMCriticalSection, DevHlp 17-77
 - SendEvent, DevHlp 17-84
- system timer rule 4-5

T

- task-time 3-1
- TCYield, DevHlp 17-89
- Terminate Environment/114H (276) 14-33
- Text Buffer Update/100H (256) 14-8
- TickCount, DevHlp 17-90
- timeout processing 8-5
- timer handler 3-6
- timer services
 - ResetTimer, DevHlp 17-76
 - SetTimer, DevHlp 17-87
 - TickCount, DevHlp 17-90
- transmit data status/return 18-42
- transmit hardware 8-4
- transmit queue 8-4
- transmit queue/return size 18-47
- transmit queue/return # characters 18-47
- Tx break status/return 18-39

U

- Unlock, DevHlp 17-91
- UnPhysToVirt, DevHlp 17-92
- UnSetIRQ, DevHlp 17-93
- using Advanced BIOS B-1

V

- Verify Font, Parallel Port IOCTL 18-116
- VerifyAccess, DevHlp 17-94
- vertical position 14-56
- Video Control IOCTL Commands (Category 3) 18-57
- video device chaining 14-2
- video device drivers, chaining 14-2
- video functions 14-5
 - DevEnable 14-5
 - Free Physical Buffer 14-28

video functions (*continued*)

- InitEnable 14-7
- Initialize Environment 14-11
- Print Screen 14-34
- Query Color Lookup Table 14-17
- Query Config Info 14-15
- Query Cursor Info 14-19
- Query DBCS Display Info 14-16
- Query Font 14-21
- Query LVB Info 14-36
- Query Mode 14-23
- Query Palette Registers 14-25
- Query Physical Buffer 14-27
- Query Variable Info 14-29
- Restore Environment 14-14
- Save Environment 14-12
- Set Color Lookup Table 14-18
- Set Cursor Info 14-20
- Set Font 14-22
- Set Mode 14-24
- Set Palette Registers 14-26
- Set Variable Info 14-31
- Terminate Environment 14-33
- Text Buffer Update 14-8
- Write TTY 14-35
- 10AH (266) 14-21
- 10BH (267) 14-22
- 10CH (268) 14-23
- 10DH (269) 14-24
- 10EH (270) 14-25
- 10FH (271) 14-26
- 100H (256) 14-8
- 101H (257) 14-11
- 102H (258) 14-12
- 103H (259) 14-14
- 104H (260) 14-15
- 105H (261) 14-16
- 106H (262) 14-17
- 107H (263) 14-18
- 108H (264) 14-19
- 109H (265) 14-20
- 110H (272) 14-27
- 111H (273) 14-28
- 112H (274) 14-29
- 113H (275) 14-31
- 114H (276) 14-33
- 115H (277) 14-34
- 116H (278) 14-35
- 117H (279) 14-36
- Video IOCTL commands (Category 3)
 - Initialize Call Vector Table, 73H 18-58
- VideoPause, DevHlp 17-95
- VirtToLin, DevHlp 17-96
- VirtToPhys, DevHlp 17-97
- VMAlloc, DevHlp 17-98
- VMFree, DevHlp 17-100
- VMGlobalToProcess, DevHlp 17-101

- VMLock, DevHlp 17-103
- VMProcessToGlobal, DevHlp 17-105
- VMSetMem, DevHlp 17-107
- VMUnlock, DevHlp 17-108
- v1.3 16-bit PPDs, running on OS/2 2.0 A-1

W

- well-behaved devices 4-5
- well-behaved monitor applications 6-19
- WRITE 8-4, 8-26
- Write TTY/116H (278) 14-35
- WRITE WITH VERIFY, strategy command 15-11
- WRITE (output), strategy command 15-11
- WRITE/multiple requests 8-26
- WRITE/ordered requests 8-26
- WRITE/queueing of requests 8-26
- WRITE/throughput 8-26
- WRITE/timeout processing 8-26
- WRITE/transmit hardware 8-26
- WRITE/transmit queue 8-26

X

- XOFF/simulate 18-17
- XON/simulate 18-18

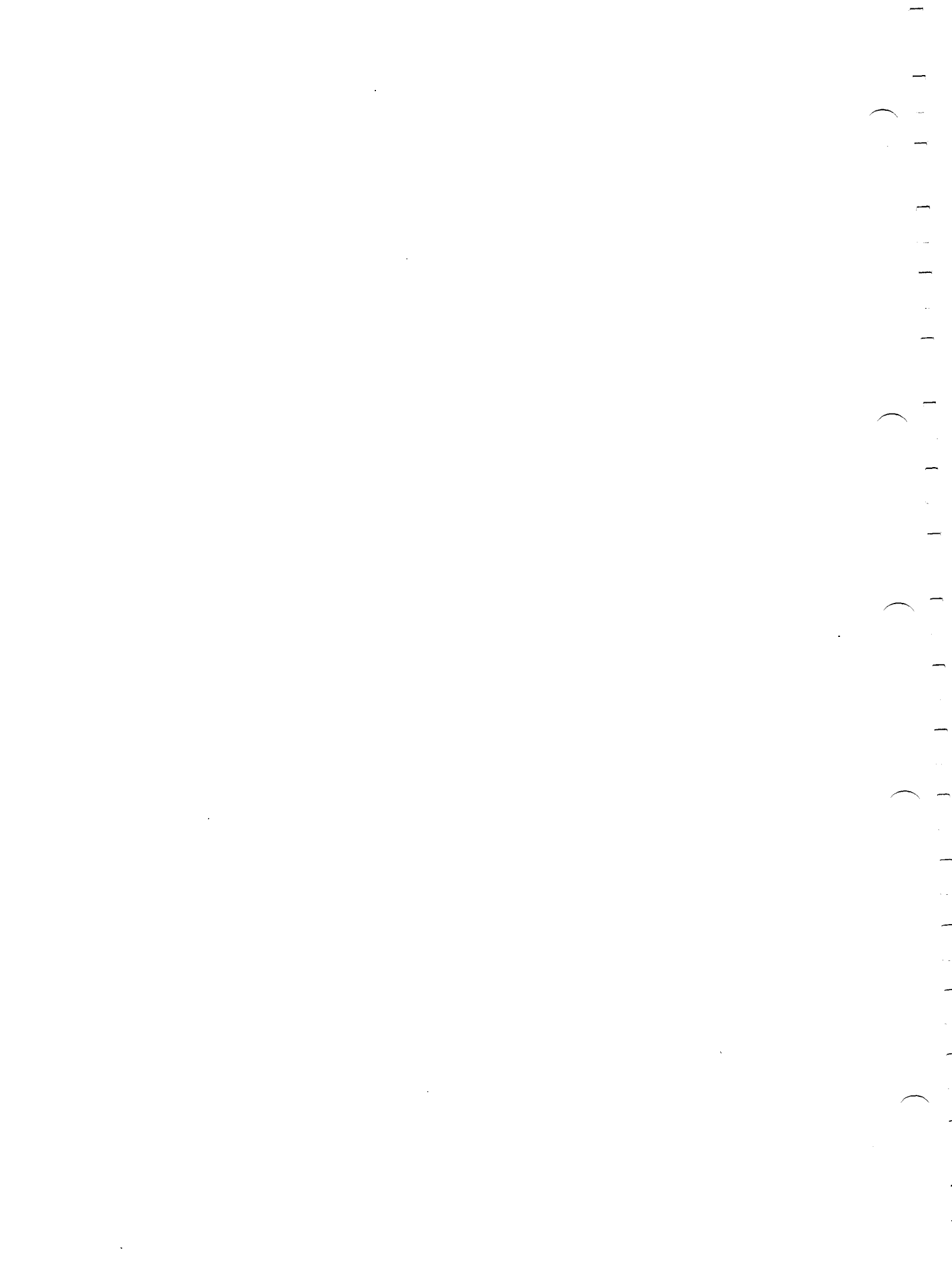
Y

- Yield, DevHlp 17-109

Numerics

- 8259 Interface, parallel port, printer 13-8







® IBM, OS/2 and Operating System/2 are
registered trademarks of
International Business Machines Corporation

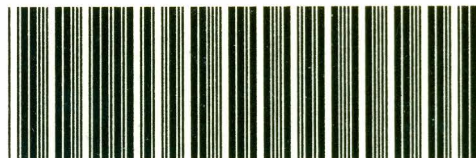


© IBM Corp. 1992

International Business
Machines Corporation

Printed in the
United States of America
All Rights Reserved

10G6266



S10G-6266-00



P10G6266